

AD-A261 404



2

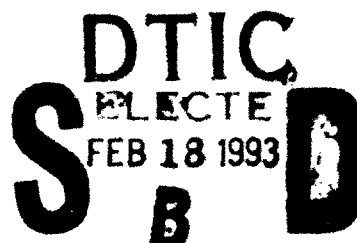
NSWCDD/MP-92/304

**PROCEEDINGS OF THE 1992 COMPLEX SYSTEMS
ENGINEERING SYNTHESIS AND ASSESSMENT
TECHNOLOGY WORKSHOP**

CUONG NGUYEN, Coordinator

UNDERWATER SYSTEMS DEPARTMENT

20-24 JULY 1992



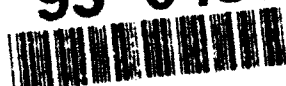
Approved for public release; distribution is unlimited.



NAVAL SURFACE WARFARE CENTER

Dahlgren, Virginia 22448-5000 • Silver Spring, Maryland 20903-5000

93-04839



NSWCDD/MP-92/304

**PROCEEDINGS OF THE 1992 COMPLEX SYSTEMS
ENGINEERING SYNTHESIS AND ASSESSMENT
TECHNOLOGY WORKSHOP**

CUONG M. NGUYEN, Coordinator

UNDERWATER SYSTEMS DEPARTMENT

20-24 JULY 1992

Approved for public release; distribution is unlimited.

NAVAL SURFACE WARFARE CENTER

Dahlgren, Virginia 22448-5000 • Silver Spring, Maryland 20903-5000

FOREWORD

1992 COMPLEX SYSTEMS ENGINEERING SYNTHESIS AND ASSESSMENT TECHNOLOGY WORKSHOP (CSESAW '92)

Mission critical computer (MCC) systems in the Department of Defense, and in particular, the Navy, are extremely large and complex, controlling a wide variety of assets operating in many unforeseeable situations. These systems have hard real-time, stringent fault tolerance and intensive security requirements. They are typically implemented on a combination of parallel and distributed architectures. In addition, these systems are generally embedded within a human organization structure and/or have human operators in the loop.

The emphasis of CSESAW (pronounced seesaw) '92 is on exploring system-level design synthesis and assessment capabilities for MCC systems. These capabilities will facilitate the development of such systems from informal system requirements, through the design phase prototyping, and into implementation and post deployment. Component products produced by these capabilities are specifications that subenvironments [e.g., hardware engineering environment (HWEE), software engineering environment (SEE), and human computer interaction engineering environment (HCIEE)] will receive. The focus of this workshop is the development and integration of these multiple technologies and the exploration of the creation of a system-level engineering discipline with support technologies to provide potential high payoff solutions to the difficult problems encountered by designers, developers, and maintainers of hard real-time MCC systems. The emphasis is on resolving system-level technology issues that cut across component boundaries, such as those associated with system behavior requirements of real time, fault tolerance, and security.

Formidable challenges await the technology developers. First, there is a need to establish strong scientific and engineering foundations with its associated technological advances in methodologies, processes, techniques, and supporting mechanizations. Since it is systems that need to be engineered, a system perspective should be followed, allowing the orchestration, interaction, and integration of the engineering of system components.

Second, the technologies and capabilities need to be integrated with the rest of the engineering process. Therefore, the capability to provide tight linkages to detailed design evaluation, systems forward engineering, and systems reengineering must be developed, ultimately providing a seamless overall engineering process. A

significant amount of effort has been put into component technologies, such as hardware, microelectronics, memory, databases, software, man-machine interface, etc. Major strides have been made in these areas in the last few years; however, the formal, systematic integration and engineering of these components into an overall system has lagged far behind. For hard real-time MCC systems with high fault tolerance and security requirements, the problem is especially acute. This is a direct result of a lack of a system-level engineering methodology.

Last, understanding the needs of the application communities and the customers is of utmost importance to the successful transition of the technologies. Demonstration of the correct technological capabilities must be based on the correct scale, context, and scope of the target applications. Correct scale implies that the technology can be applied to the problem size the application calls for. The correct context means the technology is specific (or general) enough for the application domain. The correct scope dictates that the technology addresses the nonfunctional attributes (real-time, fault tolerance, etc.) that the particular application requires.

It is in the spirit of working to meet these challenges that we welcome you to this workshop. We hope to provide in the workshop an atmosphere in which the participants, including technology developers, researchers, users, and customers, can meet, interact, and exchange ideas on relevant issues. In the near future, we hope to be able to say that this workshop was the beginning of a new generation in systems design synthesis.

This workshop would not have been possible without the hard work of many people, including the workshop, program, and advisory committees; authors; presenters of the submitted papers; panel members; workshop attendants; panel chairs; and breakout session chairs. A very warm "thank you" is extended to all. In particular, we wish to acknowledge Michael Edwards, Ngocdung Hoang, Cuong Nguyen, Michael Jenkins, Chuck Sadek, Kathy Lederer, Adrien Meskin, Dong Choi, and Janet Higgins. Finally, a special thanks goes to Elizabeth E. Wald and CDR Grace Thompson of the Office of Naval Technology for tirelessly working for and supporting the technology developments in this important area.

We hope you have a productive and enjoyable workshop!

Steven L. Howell
Assistant Workshop Chairman

Philip Q. Hwang
Workshop General Chairman

ABSTRACT

1992 COMPLEX SYSTEMS ENGINEERING SYNTHESIS AND ASSESSMENT TECHNOLOGY WORKSHOP (CSESAW '92)

The emphasis of CSESAW '92 is on exploring system-level design synthesis and assessment capabilities for mission critical computer systems. These capabilities will facilitate the development of such systems from informal system requirements, through the design phase prototyping, and into implementation and post deployment. Component products produced by these capabilities are specifications that subenvironments will receive. The focus of this workshop is the development and integration of these multiple technologies and the exploration of the creation of a system-level engineering discipline with support technologies to provide potential high payoff solutions to the difficult problems encountered by designers, developers, and maintainers of large, complex, real-time systems. The emphasis is on resolving system-level technology issues that cut across component boundaries, such as those associated with system behavior requirements of real time, fault tolerance, and security.

DTIC QUALITY INSPECTED 3

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

AGENDA

1992 COMPLEX SYSTEMS ENGINEERING SYNTHESIS AND ASSESSMENT TECHNOLOGY WORKSHOP

20-24 July 1992

Bennett Auditorium

**Naval Surface Warfare Center Dahlgren Division
White Oak Detachment (NSWCWODET)
10901 New Hampshire Avenue, Silver Spring, MD 20903-5000**

Monday, 20 July 1992

0800-0900 Registration
Bennett Auditorium Foyer, NSWCWODET

0830 Welcome—CAPT Richard Moore
NSWCWODET

Workshop Overview—Steven Howell
NSWCWODET

Real-time Dependable Systems Design I

0930 *Continuous Availability for Mission Critical Services*
F. Cristian—University of California—San Diego

1000 Break

1015 *State Restoration in Real-Time Fault-Tolerant Systems*
F. Jahanian—IBM

1045 *Fault-Tolerant Convergent Voting in Large Sparsely Connected
Networks*
R. Kieckhafer—University of Nebraska—Lincoln

1115 *Applications and Extensions of the DRB Technology for Design of
Real-Time Fault-Tolerant Distributed Computer Systems*
K. H. (Kane) Kim—University of California—Irvine

1145 Lunch

Real-time Dependable Systems Design II

- 1215 *The BEAVER Program: A Tool for Survivability Analysis of Conceptual Distributed Systems*
LT C. Whitcomb—U.S. Navy
- 1300 *Application of the Hybrid Fault Model*
M. Hugue—Allied Signal
- 1330 *Design Capture for System Dependability*
J. Zhou—Allied Signal
- 1400 *Using a POSIX Platform to Program Real-Time Concurrency and Time Fault Tolerance in Complex Systems*
V. Wolfe—University of Rhode Island
- 1430-1700 Panel: Integration of Dependability into Systems Design
Chair: Robert Goettge—Advanced Systems Technology
Members: K. H. (Kane) Kim—University of California—Irvine
Jeffrey Zhou—Allied Signal
Farnam Jahanian—IBM
Eric Brehm—Advanced Systems Technology
Michelle Hugue—Allied Signal

Tuesday, 21 July 1992

	Metrics Auditorium	Design Capture Methods Ticonderoga Room
0800	<i>Developing a Metrics Assessment Program for the SLBM Software Development Division</i> W. Farr—Naval Surface Warfare Center Dahlgren Division (NSWCDD)	<i>A Framework for the Engineering/Reengineering of Complex Systems</i> B. Blum—Johns Hopkins University/ Applied Physics Lab (JHU/APL)
0830	<i>System Design Factors</i> C. Nguyen—NSWCWODET	<i>A View to an Implementation</i> N. Hoang—NSWCWODET
0900	No presentation in this time slot	<i>The Environmental Capture View: Addressing External Factors in Capture and Analysis of Large Scale Complex System Design</i> N. Karangelen—Trident Systems
0930	Break	

1000	<i>A Method for the Assessment of System Designs</i> J. Litke-Grumman Corporate Research Center	<i>A Study for the Development of Requirements Traceability Model</i> B. Ramesh-Naval Postgraduate School
1030	<i>Methodology for Validating Software Metrics</i> N. Schneidewind-Naval Postgraduate School	<i>An Interactive Natural Interface for Formal Capture of Complex System Requirements</i> L. Hinton-Trident Systems
1100	<i>The Consolidated Experience Factory: An Approach for Instrumenting System Engineering</i> R. Vienneau-Kaman Sciences	<i>Integrated System Designer</i> M. Blanchard-Science and Technology Associates
1130	Lunch	

Integration 1

1230	<i>Strengthening the Systems/Software Engineering Interface for Real-Time Systems</i> M. Alford-Ascent Logic Corporation
1300	<i>START/ES-An Expert System Tool for System Performance and Reliability Analysis</i> R. Goettge-Advanced Systems Technologies
1330	<i>Formalizing the Transition from Specification to Design for Real-Time Systems</i> A. Gabrielian-Uniview Systems
1400	<i>Cooperative Resource Management in R-Shell</i> W. Zhao-Texas A&M
1430-1700	<p>Panel: Application of Assessment Techniques Throughout the System Development Process</p> <p>Chair: Richard Nance-Virginia Polytechnic Institute & State University (VPI&SU)</p> <p>Members: William Farr-NSWCDD Erwin Warshawski-JRS Labs José Muñoz-Naval Undersea Warfare Center (NUWC) Noah Prywes-Computer Command and Control Company (CCCC) Norman Schneidewind-Naval Postgraduate School</p>

Wednesday, 22 July 1992

	Foundations of System Engineering Auditorium	System Engineering I Ticonderoga Room
0800	<i>An Extended Event Descriptor for Real-Time Systems</i> C. Biow—University of Maryland	<i>Massively Interconnected Models for a Beam Former</i> C. Lee—Naval Postgraduate School
0830	<i>Economical Development of Complex Computer Systems</i> T. Choinski—NUWC	Analyzing Concurrent and Fault-Tolerant Software using Stochastic Reward Nets K. Trivedi—Duke University
0900	<i>Improving Safety Margins in Rate Monotone Scheduling</i> R. Menon—Texas A&M	<i>The Network Synthesis System</i> E. Warshawsky—JRS Lab
0930	<i>Stochastic Scheduling for Distributed Real-Time Systems</i> H. Moin—University of California—Santa Barbara	<i>Reliability of Redundant Arrays of Inexpensive Disks (RAID)</i> K. Trivedi—Duke University
1000-1700	Breakout Sessions	
	<u>Traceability</u> Chair: Michael Edwards—NSWCWO DET Moderator:	
	<u>Impact of Technology on Design</u> Chair: Cuong Nguyen—NSWCWO DET Moderator: Evan Lock—CCCC	

Thursday, 23 July 1992

0800 Reports on Breakout Sessions

1000 Break

Design Synthesis Methods

1030 *Rapid Prototyping*
J. Higgins—NSWCWO DET

1100 *A New Paradigm for the Design and Implementation of Large-Scale Distributed Real-Time Systems*
I. Lee—CCCC, University of Pennsylvania

1130 Lunch

	Design Synthesis Methods Auditorium	Integration II Ticonderoga Room
1230	<i>Benefits of Using Object-Oriented Methodology for Missile Guidance Processor Software Development</i> S. Lee-Governors State University	<i>Exchange of Information between Design Capture and Design Optimization Techniques: The Destination Interface Specification</i> E. Lock-CCCC
1300	<i>Embedded Computer System Requirements Methods, Analysis, and Improvement</i> S. White-Grumman Corporate Research Center	<i>Integrated System Evaluation</i> D. Choi-NSWCWO DET
1330	<i>Improving the Practice in Computer-Based Systems Engineering</i> S. White-Grumman Corporate Research Center	<i>Organizing Top-Level Systems Requirements</i> R. Jeffords-Naval Research Laboratory
1400-1700	Panel: Unification of Capture Methodologies and Representation Information Chair: Nicholas Karangelen-Trident Systems Members: Evan Lock-CCCC John Rumbut-NUWC Armen Gabrielian-Uniview Systems Bruce Blum-JHU/APL David Oliver-General Electric (GE)	

Friday, 24 July 1992

System Engineering II

0800	<i>The Component Manager: Supporting Interactive and Automated Retrieval of Real-Time Software Components</i> W. Rossak-New Jersey Institute of Technology
0830	<i>The Design of the MARUTI System</i> D. Mossé-University of Maryland
0900	<i>Lessons Learned in Modelling for Architecture Analysis</i> J. Strand-Mystech Associates
0930	Workshop Close

CONTENTS

1992 COMPLEX SYSTEMS ENGINEERING SYNTHESIS AND ASSESSMENT TECHNOLOGY WORKSHOP

	<u>Page</u>
FOREWORD	i
Phillip Hwang and Steven Howell—Naval Surface Warfare Center Dahlgren Division White Oak Detachment (NSWCWODET)	
AGENDA	v

REAL-TIME DEPENDABLE SYSTEMS DESIGN I

<i>Continuous Availability for Mission Critical Services</i>	3
F. Cristian—University of California—San Diego	
<i>State Restoration in Real-Time Fault-tolerant Systems</i>	21
F. Jahanian—IBM	
<i>Fault-Tolerant Convergent Voting in Large Sparsely Connected Networks</i>	31
R. Kieckhafer, M. Azadmanesh—University of Nebraska—Lincoln	
<i>Applications and Extensions of the DRB Technology for Design of Real-Time Fault-Tolerant Distributed Computer Systems</i>	53
K. H. (Kane) Kim—University of California—Irvine	

REAL-TIME DEPENDABLE SYSTEMS DESIGN II

<i>The BEAVER Program: A Tool for Survivability Analysis of Conceptual Distributed Systems</i>	69
LT C. Whitcomb—U.S. Navy D. Allinger—Draper Laboratory	
<i>Application of the Hybrid Fault Model</i>	93
M. Hugue—Allied Signal	
<i>Design Capture for System Dependability</i>	107
J. Zhou—Allied Signal	

	<u>Page</u>
<i>Using a POSIX Platform to Program Real-Time Concurrency and Time Fault Tolerance in Complex Systems</i>	121
J. Oblinger–Naval Undersea Warfare Center (NUWC)	
V. Wolfe–University of Rhode Island	

METRICS

<i>Developing a Metrics Assessment Program for the SLBM Software Development Division</i>	139
W. Farr, A. Ashton–Naval Surface Warfare Center Dahlgren Division (NSWCDD)	
<i>System Design Factors</i>	147
C. Nguyen, S. Howell–NSWCWODET	
<i>A Method for the Assessment of System Designs</i>	155
J. Litke–Grumman Corporate Research Center	
<i>Methodology for Validating Software Metrics</i>	171
N. Schneidewind–Naval Postgraduate School	
<i>The Consolidated Experience Factory: An Approach for Instrumenting System Engineering</i>	201
R. Vienneau–Kaman Sciences	

DESIGN CAPTURE METHODS

<i>A Framework for the Engineering/Reengineering of Complex Systems</i>	209
B. Blum–Johns Hopkins University/Applied Physics Lab (JHU/APL)	
<i>A View to an Implementation</i>	223
N. Hoang–NSWCWODET	
N. Karangelen–Trident Systems	
<i>The Environmental Capture View: Addressing External Factors in Capture and Analysis of Large Scale Complex System Design</i>	235
N. Karangelen–Trident Systems	
<i>A Study for the Development of Requirements Traceability Model</i>	249
B. Ramesh–Naval Postgraduate School	
M. Edwards–NSWCWODET	

<i>An Interactive Natural Interface for Formal Capture of Complex System Requirements</i>	263
L. Hinton, N. Karangelen-Trident Systems	
<i>Integrated System Designer</i>	271
M. Blanchard-Science and Technology Associates	

INTEGRATION I

<i>Strengthening the Systems/Software Engineering Interface for Real-Time Systems</i>	291
M. Alford-Ascent Logic Corporation	
<i>START/ES-An Expert System Tool for System Performance and Reliability Analysis</i>	303
E. Brehm, R. Goettge-Advanced Systems Technologies	
F. McCaleb-National Aeronautics and Space Administration (NASA)	
<i>Formalizing the Transition from Specification to Design for Real-Time Systems</i>	319
A. Gabrielian-Uniview Systems	
<i>Cooperative Resource Management in R-Shell</i>	327
W. Zhao, S. Natarajan-Texas A&M	

FOUNDATIONS OF SYSTEM ENGINEERING

<i>An Extended Event Descriptor for Real-Time Systems</i>	345
C. Biow, D. Rombach-University of Maryland	
<i>Economical Development of Complex Computer Systems</i>	361
T. Choinski-NUWC	
<i>Improving Safety Margins in Rate Monotone Scheduling</i>	371
R. Menon, A. Kanevsky, S. Natarajan, M. Kim-Texas A&M	
<i>Stochastic Scheduling for Distributed Real-Time Systems</i>	391
H. Moiin, P. Melliar-Smith-University of California-Santa Barbara	

	<u>Page</u>
SYSTEM ENGINEERING I	
<i>Massively Interconnected Models for a Beam Former</i>	409
C. Lee—Naval Postgraduate School	
<i>Analyzing Concurrent and Fault-Tolerant Software using Stochastic Reward Nets</i>	423
G. Ciardo, J. Muppala—Software Productivity Consortium	
K. Trivedi—Duke University	
<i>The Network Synthesis System</i>	463
E. Warshawsky—JRS Lab	
<i>Reliability of Redundant Arrays of Inexpensive Disks (RAID)</i>	473
M. Malhotra, K. Trivedi—Duke University	
DESIGN SYNTHESIS METHODS	
<i>Rapid Prototyping</i>	503
J. Higgins—NSWCWODET	
<i>A New Paradigm for the Design and Implementation of Large-Scale Distributed Real-Time Systems</i>	509
I. Lee—University of Pennsylvania	
N. Prywes—Computer Command and Control Company (CCCC)	
<i>Benefits of Using Object-Oriented Methodology for Missile Guidance Processor Software Development</i>	517
B. Hoover, S. Lee—Governors State University	
<i>Embedded Computer System Requirements Methods, Analysis, and Improvement</i>	525
S. White—Grumman Corporate Research Center	
<i>Improving the Practice in Computer-Based Systems Engineering</i>	543
S. White—Grumman Corporate Research Center	

INTEGRATION II

<i>Exchange of Information between Design Capture and Design Optimization Techniques: The Destination Interface Specification</i>	553
I. Lee, E. Lock, R. Purushothaman, M. Lee—CCCC	
<i>Integrated System Evaluation</i>	575
D. Choi—NSWCWODET	
<i>Organizing Top-Level Systems Requirements</i>	581
R. Jeffords, B. Labaw—Naval Research Laboratory	

SYSTEM ENGINEERING II

<i>The Component Manager: Supporting Interactive and Automated Retrieval of Real-Time Software Components</i>	591
W. Rossak, A. Stoyenko, L. Welch—New Jersey Institute of Technology	
<i>The Design of the MARUTI System</i>	609
D. Mossé, M. Saksena, A. Agrawala—University of Maryland	
<i>Lessons Learned in Modelling for Architecture Analysis</i>	627
J. Strand, M. Tamucci—Mystech Associates	
J. Muñoz—NUWC	

REAL-TIME DEPENDABLE SYSTEMS DESIGN I

Continuous Availability for Mission Critical Services

Flaviu Cristian

Computer Science and Engineering Department
University of California, San Diego
La Jolla, CA 92093-0114

June 11, 1992

Abstract

This paper describes a new kind of distributed system service, the Availability Management service, responsible for ensuring that the mission critical services of a distributed system remain continuously available to users despite node removals and node restarts caused by failures, maintenance and growth. The presentation stresses the main ideas behind this new service, and outlines a simple design that depends upon the existence of synchronous membership and atomic broadcast group communication services. Extensions of this initial design to deal with asynchronous group communication services are also briefly discussed.

1 Introduction

With the ever increasing dependence on computing services, their availability in the presence of failures, maintenance and horizontal growth becomes of paramount importance. In present systems, responsibility for reconfiguring a system after failures or removal of processors for maintenance rests mostly with the human system operators. Humans tend to have fairly slow reaction times, and this can result in lengthy intervals during which critical services will be unavailable. Also, humans are notorious for making mistakes, especially when under stress, and the mistakes made while attempting repair actions can lead to further failures, causing further unavailability. For example, [Gray86] reports that 42 % of the failures in the Tandem distributed systems are caused by human mistakes made during maintenance, operation and configuration.

To ensure automatic reconfiguration in the presence of failures and maximize the availability of critical services, the Advanced Automation System [CDD9], built for supporting US air traffic control in the 21st century, uses a new Availability Management service, that automatically reconfigures servers implementing critical services in the presence of processor removals caused by failures and maintenance and processor additions caused by restart, repair and horizontal growth.

The purpose of this paper is to explain in a pedagogical manner the main ideas behind this new service and outline a simple way of implementing such a service. Our presentation sacrifices the description of many of the details involved in a realistic Availability Management service design for the purpose of making the concepts on which the service and its design are based easily understandable. To this end, we focus on designing our Availability Management service on top of an easy to understand synchronous communication environment and we deal with only one kind of service availability policy. We conclude by discussing how our initial specification and design can be extended to deal with asynchronous systems subject to partitioning as well as with other kinds of service availability policies.

We begin by introducing our basic notions, system structure and assumptions and by stating the requirements that a Service Availability Management service should satisfy. A replicated implementation is then briefly described. A discussion of possible extensions concludes the paper.

2 Basic Concepts

Every computing system is built to provide services to its users. A *service* provided to a user is a system behavior as perceived by that user: a sequence of observable outputs triggered by a sequence of service operation invocations. The service specification prescribes future service outputs solely in terms of the operations being currently invoked and the current service state, where the current *service state* is the result of applying the operation invocations completed so far to the initial service state. Services that share the same set of invocable operations and the same set of potential behaviors belong to the same service *type*. In general, each service of a certain type has a current state distinct from that of other services of the same type, depending on the history of operation invocations completed so far for that service. For example, the relational ANSI SQL query and update operations together with the semantics defined for these operations in an ANSI SQL manual define the ANSI SQL relational database service type; if “employees” and “accounting” are two database services of this type, their states at a certain point in time are in general different, depending on the history of updates applied since their creation.

The operations defined for a service can only be carried out by a *service implementation* consisting of one or more servers (or objects). A *server* encapsulates private state data by a set of procedures (or methods) that provide the only way for changing and accessing the server’s state. A server is a unit of failure and growth: at any point in time a service implementation has a membership consisting of an integer number of servers. Because servers are defined to be units of failure and growth, a server cannot span several host systems that can fail independently. Service operation invocations result in server procedure executions which can cause the state of the servers implementing the service to change. Since the state of a service is a function of the states of the servers that implement it, such server state changes lead in turn to service state changes.

Note that it is vital to avoid confusing the notions of service and server (or object). In the object-oriented programming literature, the term object is often used in a confusing way to designate both what we call service and what we call server. The confusion is understandable if each service is implemented by a single object, so that there is a one to one mapping between services and objects. This was historically the case for most of the work on object-oriented programming, where issues related to fault-tolerance or replication were not a primary focus. The confusion becomes awkward when a service is implemented by several redundant servers (or objects) which are independent units of failure and growth. In this latter case, the objects that implement the service can fail and restart *without* the service users observing

any service failure or restart. Thus, when services must remain available despite the failure of some of the servers (or objects) that implement them, it becomes imperative to distinguish between services and servers.

Object oriented programming methodology requires that service users not know any details about how the state of a service is represented in terms of server states or how the service operations are implemented by the server procedures. Such implementation details are hidden from users, who need only know the externally visible, abstract service specification [Parnas72]. For example, a database service can be implemented by a single database server, by a set of distributed servers that each manages a fragment of the database state, or by a group of redundant distributed database servers that each manages a replica of the entire database. If redundant servers are used to implement a service, the user need not know what synchronization and replication policies exist among servers. The *synchronization policy* prescribes how far apart the local states of the servers can get, where the distance between the local states of two servers consists of the difference in the number of updates to the initial state applied so far by them. If the policy is *loose synchronization*, a *primary* maintains the current service state while one or more *backups* maintain past service states. A bound on the distance between loosely synchronized servers can be maintained by periodic checkpoints of the state of the primary to backups. If the policy is *close synchronization*, then the servers act as *peers* by interpreting all service requests in parallel and maintaining their internal states close to each other. The *replication policy* for a service *s* specifies how many servers should exist for *s*. For example a replication policy of 2 specifies that 2 redundant servers should be used to implement *s*. The synchronization and replication policies specified for a service constitute the *availability policy* for that service.

3 System Model and Assumptions

We consider a distributed system consisting of *nodes* linked by a *communication network*. Nodes can be uniprocessors or multiprocessors, but what is essential is that they are units of failure and growth: at any point in time a node is perceived as either correctly running (or active) or failed (not active) by another node. Servers run in the nodes of the system. If a node possesses all the physical resources needed for running a server for a certain service, it is called a (potential) *host* for that service. For example a node with enough computing power and memory that can access the disk(s) storing the "employees" database is a potential host for the "employees" database service. The set of all servers that can run in the hosts defined for a service *s* forms the *team*

of servers that can be used to implement s . With present operating systems, a server for a certain service can be started in a host node only by another process that runs in the same node. We assume the nodes have amnesia-crash failure semantics: after a crash, a node restarts in a predefined initial state independent of the inputs seen before the crash [Cris91]. We do not assume any particular network topology: it can be point-to-point or broadcast channel based.

To make the presentation of Availability Management as simple as possible, we will assume a *synchronous* [Cris91] communication network. Roughly speaking, a synchronous network enables any two active servers to communicate within a known, bounded time, so that no communication partitions are possible (a more elaborate definition and ways to implement such communication networks are given in [Cris91]). The synchronicity assumption enables us to avoid discussing issues related to potential divergence of states among redundant servers due to partitioning and the need to restrict activity to majority groups in order to prevent such divergence. We discuss later extensions to the case of asynchronous communication networks that do not guarantee a bound on the time it takes for an active server to send a message to another active server.

A synchronous network allows one to implement three group communication services that are fundamental for implementing the replicated data management needed for implementing automatic availability management: internal clock synchronization, synchronous atomic broadcast and synchronous membership.

An internal *clock synchronization* service ensures that the clocks of active nodes are synchronized within some known constant maximum deviation at any point in real time and that such synchronized clocks run within a linear envelope of real time. Protocols for achieving internal clock synchronization in synchronous communication networks are described in [CAS86], [Cris89], [HSS84], [K87], [LM85], [LL84], [S87], and [ST87].

A synchronous *atomic broadcast* service enables any member s of a team A to broadcast at any (synchronized) clock time T a message m to the group of active A members so that the following properties hold, for some time constant D . If s initiates the broadcast of m at clock time T , then at $T+D$, m is either delivered to all A members that are active or is not delivered to any active member (atomicity). All messages delivered are delivered in the same order at each active A member (order). If the sender s does not fail while broadcasting m , then all active A members deliver m at $T+D$ (termination). Protocols for implementing synchronous atomic broadcast services for point-to-point and broadcast channel based networks are given in [BD85], [CASD85] and [Cris90]. All of these protocols depend on the existence of an underlying internal

clock synchronization service.

A synchronous *membership* service enables active A members to agree in a bounded time on what member failures and recoveries happen in the history of the team. To achieve this, a membership service organizes active team members into a sequence of dynamic *groups* that exist over time, so that the following properties hold, for some time constants D and J. New groups are created only in response to failures and recoveries (stability). All members of a group agree on the membership of the group (agreement). After joining a common group, any two active A members a, b will join the same sequence of groups for as long as they stay active, so that they see all failure and recovery events that affect the team A in the same order (order). Any failure of an active A member f is detected within D clock time units, that is, it leads within at most D clock time units to the creation of a new group that includes all active A members but excludes f (bounded failure detection delay). Any recovery of a member j leads within at most J clock time units to the creation of a new group that includes j in addition to all the other active A members (bounded join delay). Several protocols for implementing a synchronous membership service in point-to-point and broadcast based networks are described in [Cris91]. The protocols depend on the existence of underlying internal clock synchronization and synchronous atomic broadcast services.

We assume that clients of a service s address service requests in a *location transparent* manner. That is, to ask for some operation o of s to be executed, a client simply passes s.o to the *request/reply transport service* available on the client's node without needing to know the location of the servers that implement s. The transport service routes the request to some subset of the servers for s and then routes the reply back. It can be connection oriented or connection-less, such as a remote procedure call service. To achieve location transparency, the request/reply transport service can make use of a *registry* service that maintains a mapping from server locations to the service they provide. When a server providing service s starts on node p, it registers with the registry service the fact that it provides service s on p. When a client on node q invokes s.o, the local request/reply transport server looks-up s in the registry and finds out that the s.o request has to be sent to node p. To ensure high availability of the registry service on which the distributed request/reply transport service depends, its implementation may consist of a team of replicated servers, that make use of the membership and atomic broadcast services described above to maintain the consistency of the replicated registry mapping in the presence of failures, recoveries and concurrently initiated updates. We assume that the request/reply transport service masks failures of s servers to clients for as long as at least one server for s remains active, so that if no timely reply to a service request s.o sent from node q to an s server is obtained, the request is automatically re-sent to some other s server that has

registered with the registry service.

4 Requirements

The goal of the Availability Management service is to enforce automatically the availability policies specified for the services offered by a distributed system to clients (without violating any constraints implied by their synchronization policies). For example if the availability policy for a service s is (loose-synchronization, 2) and the primary server fails, the Availability Management service is required to first promote the s backup to primary and then start another s backup, instead of just cold-starting another primary for s . This will minimize the time s is unavailable to clients, since it is much faster to promote a backup to primary than to cold-start a primary. Because the underlying request/reply transport service will automatically re-route client requests to the new primary after it registers as primary for s with the registry service, clients do not see a service failure: the behavior seen by them is indistinguishable from that seen when no s primary failure occurs but a request to perform some s operation is lost and re-transmitted.

To simplify the presentation of the notion of Availability Management service, we will consider that this service must automatically enforce a single availability policy for the entire set S of critical services, and that the only reason why a server for some critical service $s \in S$ can crash is the crash of its underlying node. From this presentation it should not be difficult to imagine how to deal with the cases when service implementations follow several different availability policies and when server crashes occur even if the underlying nodes do not crash. For concreteness, we will consider that the availability policy specified by the system administrator for all critical services is (loose-synchronization, 2). The reason for our choice is that primary/backup server groups are very popular commercially [Gray86] and that most of the critical services in the AAS system that we helped design are implemented by primary/backup server groups [CDD90].

If we denote the set of active system nodes by N , and the set of hosts for a service $s \in S$ by H , the Availability Management service is required to maintain a primary and a backup for s on distinct nodes as long as there exist at least two nodes in $N \cap H$ and a primary for s as long as the number of nodes in $N \cap H$ is one. If there is a backup when a primary must be started, then the backup must be promoted to primary, to minimize unavailability of s to clients (the backup does not provide service, only the primary answers to client requests). Another constraint implied by the availability

policy of s is that at no time there should exist two primaries for s . We are not concerned here with the local state check-pointing protocols followed by a primary and a backup to maintain a bound on the distance between their local states or after a new backup is started. Our view is that check-pointing is an application specific issue that is orthogonal to the system wide service Availability Management issue.

Following [Cris85], we view the Availability Management service, as exporting *two* kinds of operations to two concurrent “users”: the human administrator and the Adverse Environment. The operations that the system administrator can invoke are *start-service(s)*, *stop-service(s)*, *add-host(n)*, *remove-host(n)* and *start-node(n)*, while the Adverse Environment can invoke the *crash-node(n)* operation. Often, nodes reboot automatically after a crash, in which case the *start-node* operation is not really performed by the system administrator, but by a third concurrent “user”: the time. In other words, with automatic reboot, the passage of a certain number of time units will trigger a *start-node(n)* invocation after a *crash-node(n)* invocation by the Adverse Environment. While *start-service*, *stop-service*, *add-host* and *remove-host* are down-calls from the Administrator’s command interpreter service, the *crash-node* and *start-node* command invocations are up-calls from the underlying node membership service.

We specify our Availability Management service by first defining its abstract state and then describing the state transitions that take place in response to the above human and Adverse Environment operations.

The state of the Availability Management service is recorded by the following constants and state variables:

```

const P: Set; % the set of all nodes of the system
const S: Set; % the set of all critical services

var N: Set-of-P init {}; % set of active nodes
var H: S  $\longrightarrow$  Set-of-P init  $\lambda.\{\}$ ; % hosts for various services
var on: S  $\longrightarrow$  Boolean init  $\lambda.\text{false}$ ; % on(s)=true when  $s \in S$  is started
var primary: S  $\longrightarrow$  PU{ $\perp$ } init  $\lambda.\perp$ ; % points to node hosting primary for  $s$ 
var backup: S  $\longrightarrow$  PU{ $\perp$ } init  $\lambda.\perp$ ; % points to node hosting backup for  $s$ 

```

When there exists no primary or backup for $s \in S$, the *primary(s)* and *backup(s)* pointers have the undefined value \perp . To avoid complications related to load balancing and load shed, we will assume that all nodes in the system have distinct names that are totally ordered and that when a server for s must be started, the free host with the highest name is simply chosen to run the server for s . More realistically, the

Availability Management service will have to maintain a *load* variable that maps nodes to their load. This variable will have to be periodically updated by each node and used to select the host where a server for *s* must be started as the least loaded node (if there is a tie, take the least loaded node with the highest name). Thus, our example function for selecting the node to start a certain server among a set of potential hosts *A* is simply:

```
select-host(A:Subset-of-P) returns  $P \cup \{\perp\} \equiv$ 
  if  $A = \{\}$  then  $\perp$  else  $\max(A)$  fi;
```

The intended state transitions for the operations start-service, stop-service, add-host, remove-host, start-node and crash-node exported by the Availability Management service are as follows.

```
start-service(s:S)  $\equiv$ 
  if on(s) then inform operator "s already started"
  else on(s)  $\leftarrow$  true;
    start-servers( $N \cap H(s)$ );
  fi;
```

```
stop-service(s:S)  $\equiv$ 
  on(s)  $\leftarrow$  false;
  if backup(s)  $\neq \perp$ 
  then stop server for s on backup(s);
    backup(s)  $\leftarrow \perp$ ;
  fi;
  if primary(s)  $\neq \perp$ 
  then stop server for s on primary(s);
    primary(s)  $\leftarrow \perp$ ;
  fi;
```

```
add-hosts(h:Set-of-P, s:S)  $\equiv$ 
   $H(s) \leftarrow H(s) \cup h$ ;
  if on(s) then start-servers( $N \cap H(s)$ ) fi;
```

```
remove-hosts(h:Set-of-P, s:S)  $\equiv$ 
   $H(s) \leftarrow H(s) - h$ ;
  if backup(s)  $\in h$ 
  then stop backup server for s on backup(s);
```

```

        backup(s)  $\leftarrow$   $\perp$ ;
        start-backup(s,  $N \cap H(s)$ );
    fi;
    if primary(s)  $\in$  h
    then stop primary server on primary(s);
        primary(s)  $\leftarrow$   $\perp$ ;
        promote-backup(s,  $N \cap H(s)$ );
    fi;

start-node(n)  $\equiv$ 
     $N \leftarrow N \cup \{n\}$ ;
    for all s  $\in$  S
    do if on(s) and  $n \in H(s)$ 
        then start-servers(s,  $N \cap H(s)$ )
        fi;
    od;

crash-node(n)  $\equiv$ 
     $N \leftarrow N - \{n\}$ ;
    for all s  $\in$  S
    do if primary(s) = n then promote-backup(s,  $N \cap H(s)$ ) fi;
        if backup(s) = n
        then backup(s)  $\leftarrow$   $\perp$ ;
            start-backup(s, ( $N \cap H(s)$ ) - {primary(s)});
        fi;
    od;

```

were the start-primary, start-backup, promote-backup and start-servers state transitions are defined as follows:

```

start-primary(s:S, A:Set-of-P)  $\equiv$ 
    if primary(s) =  $\perp$  and  $A \neq \{\}$ 
    then primary(s)  $\leftarrow$  select-host(A);
        start primary server for s on primary(s)
    fi;

start-backup(s:S, A:Set-of-P)  $\equiv$ 
    if backup(s) =  $\perp$  and  $A \neq \{\}$ 
    then backup(s)  $\leftarrow$  select-host(A);
        start backup server for s on backup(s)
    fi;

```

```

promote-backup(s:S, A:Set of P)  $\equiv$ 
  if  $backup(s) \neq \perp$ 
  then promote backup server for s on  $backup(s)$ ;
     $primary(s) \leftarrow backup(s)$ ;
     $backup(s) \leftarrow \perp$ ;
    start-backup(s, A - { $primary(s)$ });
  fi;

start-servers(s:S, A:Set of P)  $\equiv$ 
  start-primary(s, A);
  if  $primary(s) \neq \perp$ 
  then start-backup(s, A - { $primary(s)$ });
  fi;

```

The above state transitions are required to be performed by the Availability Management service for s as long as at least one node is running in the system. If several events, such as failures and restarts, happen concurrently at about the same time, then the corresponding state transitions are required to occur in some serial order. Moreover, assuming that the start of a primary or backup server for service s takes a bounded amount of time, each transition is required to take a bounded amount of time. In effect, if an Availability Management service satisfying the above requirements exists in a system, it ensures the *continuous availability* of a service s to clients for as long as there exists at least one active node that can host s , despite any number of possibly concurrent node failures and joins.

5 Replicated Design

To ensure that the Availability Management service is itself available as long as at least one node is active, the constants and variables introduced in the previous section are *replicated* on all nodes. Thus, the team of Availability Management servers is hosted by the set P of all¹ nodes of the system. These replicated variables are managed at any point in time by a group of active, closely synchronized Availability Management servers. These servers are initialized after the underlying clock synchronization, node membership and atomic broadcast services are initialized at node startup. While

¹For a hierarchical approach to Availability Management in large systems, see [CDD90].

servers for these services exist at any node, the Administrator Command Interpreter service is implemented by a server running in only one of the nodes (if that node fails, the administrator can login on another node). When joining the group of active Availability Management servers, a newly started server follows the state initialization protocol described in [Cris91], where the new server gets an older value of the service state, monitors all updates to that past state until its reception and then applies these updates to the old state to get an up-to-date state. We will not repeat the description of that join protocol here. Each Availability Management server has access to the identity of its underlying node by invoking a predefined function *myid*.

The design to be described *depends* directly on the membership and atomic broadcast services described in section 3: any update to a replicated state variable is either a result of an atomic broadcast or of a membership change notification that appears to the replicated Availability Managers as an atomic broadcast (for more details see [Cris91]). Because all updates are received *in the same order* [CASD85], [Cris91] at all active Availability Managers, after an Availability Manager *j* joins the group of active Availability Managers, its local state variables *N*, *H*, *on*, *primary*, *backup* will go through the *same sequence* of values as the local variables of any other Availability Manager *m* that was already joined when *j* joined, and this will hold true until *j* or *m* fail. Thus, when any two members of the group of active Availability Managers learn about the same event, such as an administrator command invocation or a change in the membership of active nodes, they have identical local states, so they reach identical decisions about what has to be done. For example after a failure of a node hosting the primary server for *s*, all Availability Managers decide that the manager running in the *backup(s)* node which hosts the backup server for *s*, will have to promote it to primary and that the manager running in the node with the highest name in the set $N \cap H(s) - \{backup(s)\}$, say *n*, will have to start a new backup for *s*. So, all Availability Managers update their state according to these decisions and they all ask themselves whether they are the ones running in the *backup(s)* and *n* nodes by evaluating the $myid = backup(s)$ and $myid = \max(N \cap H(s) - \{backup(s)\})$ expressions, respectively. The managers running in the nodes where these expressions evaluate to true then do the "real" work by locally promoting the backup server for *s* and by locally starting a backup for *s*, respectively. When more realistic select-host functions are used instead of our simple select-host example, it is crucial that the value of an invocation of select-host depend only upon the replicated state variables maintained by each Availability Manager, so that any invocation yields the same result when invoked in response to the same event at any two members of the group of active Availability Managers.

After completing the state initialization protocol described in [Cris91] that initializes

the N , H , on , $primary$, $backup$ variables, an Availability Manager enters an infinite loop inside which it waits for the following event types: an upcall from the membership service that is a notification of a change in the membership of active nodes (and hence, in the group of active Availability Managers), an upcall from the atomic broadcast service telling about an update to the replicated state variables, or a downcall from the Command Interpreter running in the same node, that informs the server about a command issued by the system administrator. The code that implements the reactions to these events is atomic with respect to synchronization (for simplicity we do not deal explicitly with synchronization issues related to making the parallel interpretation of these events serializable).

task Availability-Manager \equiv

```

const P,S: Set;
var N: Set-of-P init {};
var H: S  $\rightarrow$  Set-of-P: init  $\lambda.$ { };
var on: S  $\rightarrow$  Boolean init  $\lambda.$ false;
var primary: S  $\rightarrow$  PU{ $\perp$ } init  $\lambda.$  $\perp$ ;
var backup: S  $\rightarrow$  PU{ $\perp$ } init  $\lambda.$  $\perp$ ;

initialize(N, H, on, primary, backup);

loop

when receive-from-administrator(command):
  case command of:
    start-service(s): if on(s)
      then send-to-administrator("already started",s)
      else if  $N \cap H(s) = \{\}$ 
        then send-to-administrator("no active hosts for",s)
        else atomically-broadcast("start-service",s)
      fi;
    fi;
  stop-service(s): if on(s)
    then atomically-broadcast("stop-service",s)
    else send-to-administrator("already-stopped",s)
    fi;
  add-hosts(h,s): if  $h \in H(s)$ 
    then send-to-administrator("already-hosts",h,s)
    else atomically-broadcast("add-hosts",h,s)
    fi;
  remove-hosts(h,s): if  $h \cap H(s) \neq \{\}$ 

```

```

        then atomically-broadcast("remove-host",  $h \cap H(s)$ , s)
        else send-to-administrator("not-hosts", h, s)
        fi;
    endcase;

when receive-atomic-broadcast(message) from p:
    case message of:
        ("start-service", s):  $on(s) \leftarrow true$ ;
            Start-Servers( $N \cap H(s)$ );
        ("stop-service", s):  $on(s) \leftarrow false$ ;
            if  $backup(s) \neq \perp$ 
            then if  $backup(s) = myid$ 
                then locally stop backup server for s;
            fi;
             $backup(s) \leftarrow \perp$ ;
            fi;
            if  $primary(s) \neq \perp$ 
            then if  $primary(s) = myid$ 
                then locally stop primary server for s;
            fi;
             $primary(s) \leftarrow \perp$ ;
            fi;
        ("add-hosts", h, s):  $H(s) \leftarrow H(s) \cup h$ ;
            if  $on(s)$  then Start-Servers( $N \cap H(s)$ ) fi;
        ("remove-hosts", h, s):  $H(s) \leftarrow H(s) - h$ ;
            if  $backup(s) \in h$ 
            then if  $backup(s) = myid$ 
                then locally stop backup server for s;
            fi;
             $backup(s) \leftarrow \perp$ ;
            Start-Backup(s,  $N \cap H(s)$ );
            fi;
            if  $primary(s) \in h$ 
            then if  $primary(s) = myid$ 
                then locally stop primary server for s;
            fi;
             $primary(s) \leftarrow \perp$ ;
            Promote-Backup(s,  $N \cap H(s)$ );
            fi;
    endcase;

when receive-membership-notification(change, n):

```

```

case change of:
  "join":  $N \leftarrow N \cup \{n\}$ ;
    for all  $s \in S$ 
      do if  $on(s)$  and  $n \in H(s)$ 
        then Start-Servers( $s, N \cap H(s)$ )
        fi;
      od;
  "crash":  $N \leftarrow N - \{n\}$ ;
    for all  $s \in S$ 
      do if  $primary(s)=n$  then Promote-Backup( $s, N \cap H(s)$ ) fi;
        if  $backup(s)=n$ 
          then  $backup(s) \leftarrow \perp$ ;
            Start-Backup( $s, (N \cap H(s)) - \{primary(s)\}$ );
          fi;
        od;
    endcase;
endloop;

```

The Start-Primary, Start-Backup, Promote-Backup and Start-Servers procedures invoked by the implementation implement in a decentralized manner the abstract state transitions defined by start-primary, start-backup, promote-backup, and start-servers of the previous section, respectively.

```

procedure Start-Primary( $s:S, A: \text{Set-of-P}$ );
  if  $primary(s)=\perp$  and  $A \neq \{\}$ 
  then  $primary(s) \leftarrow \text{select-host}(A)$ ;
    if  $primary(s)=\text{myid}$ 
    then locally start primary server for  $s$ 
    fi;
  fi;

```

```

procedure Start-Backup( $s:S, A: \text{Set-of-P}$ );
  if  $backup(s)=\perp$  and  $A \neq \{\}$ 
  then  $backup(s) \leftarrow \text{select-host}(A)$ ;
    if  $backup(s)=\text{myid}$ 
    then locally start backup server for  $s$ 
    fi;
  fi;

```

```

procedure Promote-Backup(s:S, A:Set-of-P);
  if backup(s)  $\neq \perp$ 
  then   if backup(s)=myid
         then locally promote backup server for s
         fi;
         primary(s)  $\leftarrow$  backup(s);
         backup(s)  $\leftarrow \perp$ ;
         Start-Backup(s,A - {primary(s)});
  fi;

procedure Start-Servers(s:S, A:Set-of-P);
  Start-Primary(s,A);
  if primary(s)  $\neq \perp$ 
  then Start-Backup(s,A - {primary(s)})
  fi;

```

6 Extensions

An extension of the above design that would deal with several services with distinct availability policies should pose no difficulty at this point. It is sufficient for this purpose to keep track for each service S of its availability policy and ensure that it is automatically enforced along the lines of the previously sketched Availability Management service design. Other extensions that would allow the administrator to tailor the reaction of the Availability Management service to the particular needs of a system installation by letting the administrator define the reaction that ought to take place in response to each type of event for each kind of declared availability policy in a specialized high level programming language are also quite straightforward. An important extension of the simple design presented here deals with saving the state of the Availability Management service on non-volatile storage, so as to enable a quick restart after a total system failure (possibly due to a general power failure). This is a non-straightforward problem if one wants to solve it right. We leave it to the reader to imagine various possible solutions and analyse their relative merits and drawbacks.

In the remainder of this section we limit ourselves to discuss possible extensions to the case of asynchronous communication networks that do not guarantee any bound on the time needed to communicate between two active nodes. The major difficulty in

such networks is that a process p cannot distinguish between being partitioned from another process q and a failure of q . This leads to the possibility that the members of a team A join distinct groups at the same time and see different sequences of updates to their local states. To prevent such divergence it is sufficient to let updates proceed only in majority groups. It is possible to design membership and atomic broadcast protocols that will let any two active team members that *continuously* join majority groups see the same sequence of global state updates, including node restarts and failures despite unbounded communication delays. The main drawbacks of such a solution are that no availability management will happen when less than a majority of nodes are active and that there will be no bounds on the time it takes to react to events such as administrator commands and node membership changes. Another alternative design could be based on a leader that orders all the events happening in the system by acting as a funnel for them. The hard problem there will be to elect a new leader upon failure of the old leader so as to ensure that at no point in real-time there exist two leaders that could take conflicting actions. A leader based solution will of course share the drawbacks inherent to any solution based on asynchronous communication: need for majority presence and no bounds on reaction times.

7 References

- [BD85] O. Babaoglu, R. Drumond: Streets of Byzantium: network architectures for fast reliable broadcasts, IEEE Tr. on Software Eng., Vol. SE-11, No. 6, 1985.
- [CASD85] F. Cristian, H. Aghili, R. Strong, D. Dolev: Atomic broadcast: from simple message diffusion to Byzantine Agreement, 15th Int. Symp. on Fault-tolerant Computing, 1985.
- [CAS86] F. Cristian, H. Aghili, R. Strong: Approximate clock synchronization despite omission and performance failures and processor joins, 6th Int. Symp. on Fault-tolerant Computing, 1986.
- [CDD90] F. Cristian, J. Dehn, B. Dancey: Fault-tolerance in the Advanced Automation System, 20th Int. Symp. on Fault-tolerant Computing, 1990.
- [Cris85] F. Cristian: A rigorous approach to fault-tolerant programming, IEEE Tr. on Software Engineering, Vol. 11, No. 1, 1985.
- [Cris89] F. Cristian: Probabilistic clock synchronization, Distributed Computing, Vol. 3, pp. 146-158, 1989.

- [Cris90] F. Cristian: Synchronous atomic broadcast for redundant broadcast channels, *Journal of Real-time Systems*, Vol. 2, pp. 195-212, 1990.
- [Cris91] F. Cristian: Reaching agreement on processor-group membership in synchronous distributed systems, *Distributed Computing*, Vol. 4, 1991, pp. 175-187.
- [Crist91] F. Cristian: Understanding Fault-tolerant Distributed Systems, *Communications of the ACM*, Vol. 34, No. 2, Feb 1991.
- [Gray86] J. Gray: Why do computers stop and what can be done about it?, 5th Symposium on Reliability in Distributed Software and Database Systems, 1986.
- [HSS84] J. Halpern, B. Simons, R. Strong: Fault-tolerant clock synchronization, *Proc. 3rd ACM PODS*, 1984.
- [K87] H. Kopetz: Clock synchronization in distributed real-time systems, *IEEE Tr. on Computers*, Vol. C-36, No. 8, 1987.
- [LM85] L. Lamport, M/ Melliar-Smith: Synchronizing clocks in the presence of faults, *Journal of the ACM*, Vol. 32, No. 1, 1985.
- [LL84] J. Lundelius, N. Lynch: A new fault-tolerant algorithm for clock synchronization, *Proc. 3rd ACM PODS*, 1984.
- [Parnas72] D. Parnas: A technique for software module specification with examples, *Comm. of the ACM*, Vol. 15, No. 5, 1972.
- [S87] F. Schneider: Understanding protocols for Byzantine clock synchronization, *TR Cornell*, 1987.
- [ST87] T. Shrikanth, S. Toueg: Optimal clock synchronization, *Journal of the ACM*, Vol. 34, No. 3, 1987.

State Restoration in Real-Time Fault-Tolerant Systems

Farnam Jahanian

IBM T. J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598

e-mail: farnam@watson.ibm.com

EXTENDED ABSTRACT

1 Motivation

With the increasing reliance on digital computers in embedded systems, the need for dependable systems that deliver correct results in a timely manner has become more crucial. Large-scale embedded systems are being built in diverse applications such as avionics, air traffic control, manufacturing, and patient monitoring. These systems often have strict availability and timing requirements that affect one another in subtle ways. For example, availability requirements are often enforced as timing constraints on certain tasks in the system. Alternatively, missing a deadline on a critical task in a real-time system may result in a system failure. As real-time fault-tolerant applications become more sophisticated, the software design and development process has become increasingly more complex. This paper argues that the traditional approaches for providing fault-tolerance in asynchronous distributed systems is not necessarily appropriate for time-critical applications.

The motivation for this work is based on two observations:

1. the characterization of design methodologies for fault-tolerant systems based on redundancy in space or redundancy in time is inadequate for real-time systems; and
2. establishing a global consistent system state based on the causal order of messages among cooperating processes does not consider the temporal consistency requirements imposed on the data in a system.

2 System State

Fault-tolerance can be defined informally as the ability of a system to provide a service in a timely manner even in the presence of failures. A common approach for building fault-tolerant systems is to replicate servers that fail independently. The main strategies for structuring fault-tolerant servers are passive and active replication. In passive replication schemes [4, 1], the system state is maintained by a primary and one

or more backup processes. The primary checkpoints its local state to the backups such that a backup can take over upon detecting a failure of the primary. In active replication schemes [6, 13, 2], also known as the state machine approach, a collection of identical server processes maintain replicated copies of the system state. Updates are applied atomically to all the replicas such that after detecting the failure of a process, the remaining processes continue the service.

In a distributed environment, one can view the system as a collection of cooperating processes. The global system state is the aggregate set of the local states of the cooperating processes. A widely-studied approach is to establish consistent global system states as a computation progresses and to roll back to an earlier system state when a failure is detected. A consistent global system state is defined to be a state that is reachable from the initial state. Numerous checkpointing/logging-based schemes for establishing a global system state in a distributed environment have been proposed in the past, e.g., [3, 8, 14]. In these approaches, each process checkpoints its state locally, and the messages between processes are logged synchronously or asynchronously. Upon detecting a failure, a global system state is established by a rollback to an earlier point in the computation that could have been reachable from the initial system state.

It has been argued in the past that the real-time and fault-tolerance requirements of a system are not orthogonal. The difficulty in applying traditional approaches for providing fault-tolerance to real-time systems is that time is not explicitly considered in defining a *consistent system state*. In particular, several distinguishing characteristics of real-time systems must be considered:

1. *Timing Constraints*: the correctness of a computation is dependent not only on the correctness of its results, but also on meeting stringent timing requirements.
2. *Perishable Data*: the data in these systems are perishable in the sense that the usefulness of a data item decreases with the passage of time.
3. *Weaker Consistency Constraints*: the semantics of real-time data allows the exploitation of weaker consistency requirements than the causal or total order on the operations in a system.
4. *Redundancy in Data Semantics*: the characterization of design methodologies based on redundancy in space or redundancy in time is inadequate since the semantics of certain data items allows a stale or approximate data value to be used.

Before presenting alternative models for specifying consistent system states of real-time processes, we will elaborate on the above points. We use two examples to illustrate why a different definition of a system state is more appropriate for real-time system.

Whether the primary motivation for fault-tolerance is to ensure data integrity or to mask failures at run-time, the notion of a consistent system state after a failure defines the correctness criteria for different approaches. The precise definition of a consistent state in a real-time system is complicated by one crucial factor: a system state in time-critical applications changes by the passage of time. This is a key difference from asynchronous systems in which *time* is not considered explicitly in defining a system state. This has several important implications: First, redundancy management must be predictable; meeting stringent timing constraints and achieving fault-tolerance requirements may be contradictory goals in some cases. Second, restoring a system state (by rolling backward or forward) must satisfy certain timing properties imposed on the data in the system. Since usefulness of real-time data diminishes with the passage of time, the definition of a consistent system state must include the temporal relationship between data objects. Third, the ordering constraints, such as Lamport's happened-before relation or the total order guaranteed by atomic multicast, may be weakened in managing replicated data in real-time systems. Finally, the inherent

non-determinism of time-critical systems can be exploited in developing new fault-tolerance strategies. The imprecise computation technique, for example, exploits data semantics in obtaining timely but lesser quality results in iteratively improving calculations.

A real-time system may fail to function correctly either because of errors in its hardware and/or software or because of not responding in time to meet the timing requirements that are usually imposed by its "environment." Hence, a real-time system can be viewed as one that must deliver the expected service in a timely manner even in the presence of faults. A missed deadline can be potentially as disastrous as a system crash or an incorrect behavior of a critical task, e.g., a digital control system may lose stability. Since the logical correctness of a system may be dependent on the timing correctness of other components, the task of separating logical correctness from timing correctness may be very difficult. Furthermore, timeliness and fault-tolerance requirements could pull each other in opposite directions. For example, checkpointing a system state and complex recovery mechanisms will enhance fault-tolerance but may increase the probability of missing a deadline. Hence, one must explicitly consider timing requirements when defining a consistent system state. The following example illustrates this point.

Example 1: Airspace Control

Consider an airplane that is moving from airspace A to an adjacent airspace B.¹ Different air traffic controllers are responsible for each airspace. As the airplane is moving from airspace A to airspace B, the control must be passed from one controller system to the other. Two data objects, O_A and O_B , reflect which controller is responsible for the airplane. If $O_A = 1$, controller A is in charge of the airplane. If $O_A = 0$, the controller A is not responsible for the airplane. Initially, $O_A = 1 \wedge O_B = 0$. The hand-off must take place as the airplane is moving from one airspace to the other. A safety property of the system is that there should be a maximum time interval of 500ms during which both data objects are zero, i.e., neither controller is responsible for the airplane. Suppose a process P_1 updates both O_A and O_B , and P_2 and P_3 are the displaying processes for controllers A and B respectively. We use the notation $w(O)$ and $r(O)$ to denote a write operation and a read operation to an object O , respectively.

$P_1 : w(O_A), w(O_B)$

$P_2 : r(O_A), w(display_A)$

$P_3 : r(O_B), w(display_B)$

If the above safety property (or time constraint) is not imposed on the system, any interleaved execution of the above operations is acceptable. Consider the following execution sequence:

$w(O_A), r(O_A), r(O_B), w(O_B), \dots$

If the two reads are separated by more than 500ms, the safety property is violated. Thus, the correct relative ordering of operations does not necessarily ensure temporal correctness. Hence, other constraints must be imposed to ensure this performance requirement. In this example, all operations in P_1 must be performed within 500ms. \square

Fault-tolerance techniques based on checkpointing and message logging ensure that after a failure, a distributed computation recovers to a global state which is reachable from its initial state. There are several problems in applying this approach to real-time systems: First, since a real-time process may include time explicitly in its local state, the definition of a consistent global system state based on partial (or causal) ordering of messages may not be appropriate. For example, consider the data repository in a flight control

¹This is a variation of an example in [10].

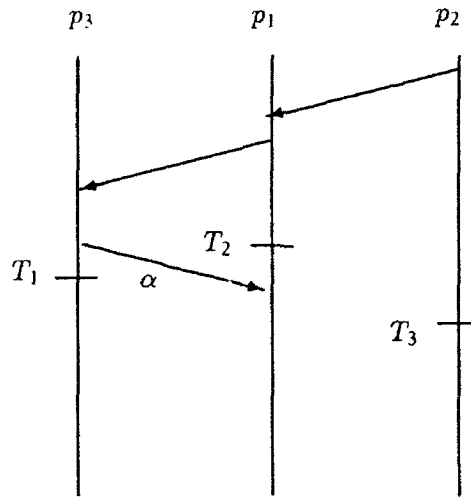


Figure 1: A Distributed Computation

system. The decision to delay or to land an aircraft is based on the "current" position and status of the aircrafts being monitored. The system state in the data repository is consistent if the timestamps of different data items representing aircraft positions are within an acceptable tolerance. Second, restoring the system to an earlier state may be unnecessary or even incorrect in these applications. In certain cases, a process may resume its execution at a predefined state and obtain its input directly from a sensor after a failure. Third, since a timing constraint may be imposed on the execution of a process (or a collection of processes), a complex recovery mechanism and resuming execution in an earlier state may result in missing the deadline.

Example 2: Distributed Computation

Figure 1 illustrates three cooperating processes P_1 , P_2 and P_3 with the corresponding checkpoints S_1 , S_2 and S_3 . The messages labeled α from process P_1 to P_2 crosses the recovery line established by the checkpoints. In an asynchronous environment, to establish a consistent cut, message α is logged synchronously or asynchronously by the sender or the receiver. If the processes in Figure 1 are real-time processes, several other alternatives for establishing a consistent system state may be possible. If the state variable v updated by α can be extrapolated from its previous values, then it may be unnecessary to log the message to establish a consistent system state. Alternatively, if process P_2 is a periodic process and the previous value of v (prior to the checkpoint at S_2) is within a predefined distance from the new value of v , this previous value of v can be used in case the process suffers a failure after the checkpoint T_2 and before the subsequent checkpoint. Another alternative may be to take checkpoints based on absolute time if the processor clocks are synchronized within a known value ϵ . For example, as shown in Figure 2, if each process takes a checkpoint at the local time T , then a recovery interval $[T - \epsilon, T + \epsilon]$ can be established. This recovery interval can be used to define a consistent global state to which the system can be restored after a failure.

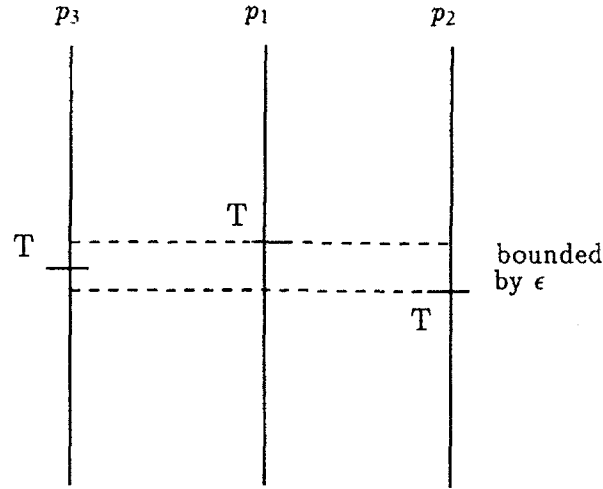


Figure 2: A Recovery Interval

3 Real-Time Models to Support Fault-Tolerance

In this section, we propose two models for defining a system state in a real-time computation. The models introduce two complementary approaches to providing fault-tolerance in real-time systems. Both approaches incorporate the notion of time into the definition of a consistent system state. Before presenting the models, we examine briefly a classification of data dependencies in a real-time computation:

- Temporal Dependency: A collection of data items whose timestamps must be within a predefined value ϵ of each other in a system state [10].
- Value Dependency: A collection of data items whose values must be within a predefined tolerance δ in a system state.
- Causal Dependency: Existence of a data item is dependent on the existence of another data item.

3.1 Server State

In a real-time system, it is important to use the values of data objects that have existed at approximately the same time. For example, an air traffic controller monitoring the positions of several aircrafts must view the coordinates that are taken within a very short interval. Hence, a set of temporal constraints must be enforced on the data objects in a system. These temporal constraints must be considered when defining a consistent system state in a real-time environment. Consequently, a system state restored after a failure must satisfy these temporal constraints. A crucial observation is that the decision on when to update a backup copy is often driven by the staleness of the data object rather than the relative (causal) order of message exchanges between processes.

In this model, a real-time system is seen as a collection of services provided in a timely and dependable manner. Each service is provided by a set of replicated servers running on multiple processors. (The

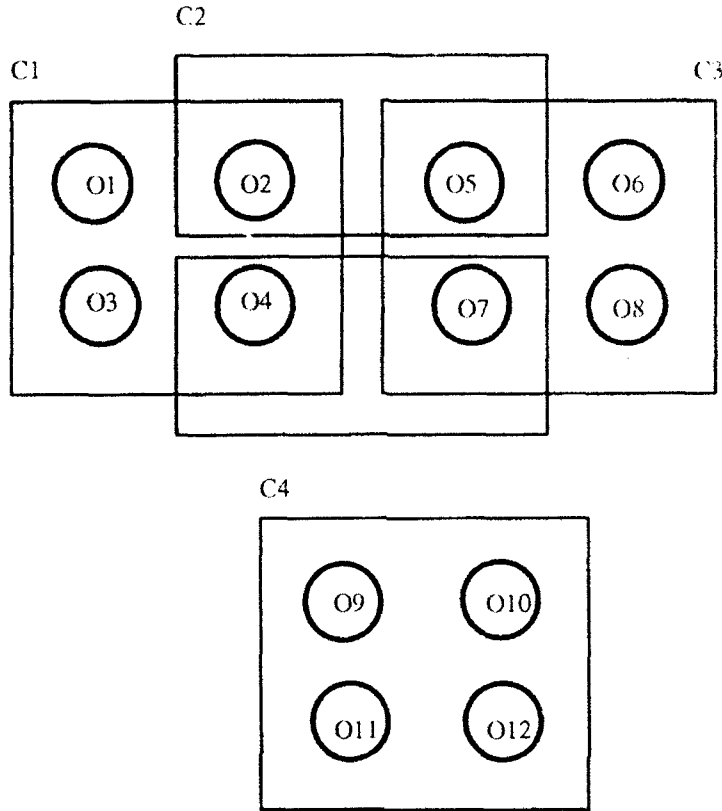


Figure 3: A Server State: Objects and Constraints

replication strategy for each service can be based on active or passive replication.) We define the state of a system as the collective states of the underlying servers. A server state, in turn, is defined by the set of objects (including the values and timestamps) internal to that server. This is where we depart from traditional approaches to defining a system state. A consistent state of a server is defined by the set of the temporal and value constraints imposed on the objects in the server. In other words, a consistent server state is the temporally correct snapshot of the objects maintained by the server. In a automated process control system, for example, the algorithms to monitor and control an external device are executed periodically. The result from the execution of an algorithm can be updated on a backup server to tolerate against the primary failure. The state of this server is the set of input and output values during each iteration of the program execution. This is the state that must be restored (in a timely fashion) if the primary server fails.

Figure 3 illustrates a server state. The objects in a server are denoted by circles; the constraints imposed on collections of objects are denoted by squares. An update to an object or the passage of time must preserve the constraints imposed on the objects. A definition of a server state based on the notions of temporal and value dependencies is given below.

Definition:

A *server state* is defined to be a ordered collection of objects (O_1, O_2, \dots, O_n) with $v(O_i)$ and $t(O_i)$ denoting the value and timestamp of object O_i .

Suppose the current time is denoted by t . A server state is *consistent* if a set of constraints are satisfied

$$\forall i, j \ |t(O_i) - t(O_j)| < \epsilon_{ij}(T)$$

$$\forall i, j \ |v(O_i) - v(O_j)| \leq \delta_{ij}(T)$$

The above definition requires a consistent system state to satisfy a set of temporal and value dependency constraints. These constraints are explicitly imposed on pairs of objects that collectively form a server state. After detecting a failure, the server must be restored to a state that satisfies the temporal and value dependencies. The parameter T indicates that ϵ and δ are not necessarily constants. For example, the value of ϵ can itself be a function of the distance between the timestamps of the pair of objects and the length of time since the minimum of the two timestamps.

The above definition for a consistent server state considers only the last update to an object, i.e., the most recent value and timestamp for an object. In certain cases, a bounded history of recent updates to an object can be exploited to define a consistent server state. For example, consider a task that periodically updates a state variable based on the calculation in the previous invocation of the task and the reading of a new value from a sensor. A backup processor will execute this task periodically if the primary fails. It is unnecessary to send the update from each iteration to the backup if an approximate value can be extrapolated from earlier iterations that have already been logged on the backup. The above definition for a consistent state can be extended to include semantic information about the history of recent updates to an object. Suppose $O_i[-k]$ denotes the k th most recent update to an object O_i . The following constraint imposes lower and upper bounds on the value of the last update based on the k th most recent updates.

$$f_i(k, T) \leq v(O_i[-1]) \leq g_i(k, T)$$

The functions f and g establish the lower and upper bounds on an acceptable value of that can be restored after a failure. The following simple constraint imposes lower and upper bounds on the difference between the last two updates:

$$f_i(T) \leq |v(O_i[-1]) - v(O_i[-2])| \leq g_i(T)$$

If the second most recent update to O_i has been sent to a backup processor, the above constraint can be used to determine whether the most recent update must be sent to the backup processor as well.

3.2 State of Cooperating Processes

In this model, a real-time system is seen as a set of cooperating (perhaps periodic) processes that communicate by exchanging messages. Processes go through internal state transitions due to internal computation, receipt of a message or passage of time. The global system state can be characterized by the set of local process states and the messages that have been sent by the sender but not applied to the local state. Instead of viewing a global state as a consistent cut of states of processes at some logical instant of time, such as [3, 5, 8, 9, 12, 14], we define a system state at an instant of real-time in the same spirit as in [7, 11, 15]. We assume that the processor clocks in the system are approximately synchronized within a known deviation ϵ .

Definition:

Suppose \mathbf{P} is a real-time distributed system consisting of a set of n processes p_1, p_2, \dots, p_n . The global

system state at time t , denoted by $S(t)$, is defined as

a) $\langle s_1(t), s_2(t), \dots, s_n(t) \rangle$, such that $s_i(t)$ is the local state of the process p_i at time t_i where $t_i = \min(t, \tau_i)$ and τ_i is the time of the receipt of the first message by p_i that was sent by another process p_j after p_i reached its local state $s_j(t)$.

b) $m_{i,j}(t)$, $\forall 1 \leq i, j \leq n$, such that $m_{i,j}(t)$ is the set of fifo messages from p_i to p_j timestamped by p_i before its local time t_i and not received by p_j until after its local state $s_j(t)$.

Part (a) in the above definition refers to the collection of local process states and part (b) covers the logical message queue between each pair of processes.

The above definition established a recovery interval to which a system can be restored after a failure. However, it still attempts to preserve the causal order when establishing a consistent system state. It is possible to relax the above definition such that a weaker notion of a system state can be obtained. One possible approach is to enforce constraints similar to those in section 3.1 to determine whether an update (a message) from a sender should be logged during the recovery interval.

4 Concluding Remarks

As embedded real-time systems become more sophisticated, the ability of the system to provide dependable and timely service becomes critical. The focus of this extended abstract was on alternative models for defining a system state in a real-time computation. It was argued that the traditional approaches to fault-tolerance in asynchronous systems are not suitable for a real-time environment. Recovering a system to a consistent state after a failure must consider the timing requirements imposed on the system. The two models that were presented in this abstract consider time explicitly in defining a consistent system state. In one model, a system state consists of a collection of temporally related objects. A system failure would require restoration of the system to a state in which the temporal and value dependency requirements among objects are satisfied. In the second model, a system state is defined at an absolute time. Due to the approximately synchronized processor clocks, a recovery interval is established to which a system can be restored after a system failure.

References

- [1] A. Bhide, E. N. Elnozah, and S. P. Morgan. A highly available network server. *USENIX*, pages 199-205, 1991.
- [2] K. P. Birman. The process group approach to reliable distributed computing. Technical Report TR 91-1216, Department of Computer Science, Cornell University, July 1991.
- [3] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault-tolerance. In *Proc. of 9th ACM Symposium on Operating Systems Principles*, pages 90-99, October 1983.
- [4] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. Primary-backup protocols: Lower bounds and optimal implementations. Technical Report TR 92-1265, Department of Computer Science, Cornell University, January 1992.
- [5] M. Chandy and L. Lamport. Distributed snapshot: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63-75, February 1985.

- [6] F. Cristian and R. Dancey. Fault-tolerance in the advanced automation system. Technical Report RJ7424, IBM Research Laboratory, San Jose, April 1990.
- [7] F. Cristian and F. Jahanian. A timestamp-based checkpointing protocol for long-lived distributed computations. In *9th Symposium on Reliable Distributed Systems*, Pisa, Italy, September 1991.
- [8] D.B. Johnson. Distributed system fault tolerance using message logging and checkpointing. In *Technical Report COMP TR89-101*. Department of Computer Science, Rice University, December 1989.
- [9] K. Kim. Approaches to mechanization of conversation scheme based on monitors. *IEEE Transactions on Software Engineering*, SE-12(5), May 1982.
- [10] K-J Lin, F. Jahanian, A. Jhingran, and C. D. Locke. A model of hard real-time transaction systems. Technical Report RC 17515, IBM T.J. Watson Research Center, January 1992.
- [11] P. Ramanathan and K.G. Shin. Use of common time base for checkpointing and rollback recovery in a distributed real-time system. *submitted for publication*, 1991.
- [12] D.L. Russell. State restoration in systems of communicating processes. *IEEE Trans. on SE*, SE-6(2), March 1980.
- [13] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4), December 1990.
- [14] R.E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.
- [15] Z. Tong, R.Y. Kain, and W.T. Tsai. A low overhead checkpointing and rollback recovery scheme for distributed systems. In *Symposium on Reliable Distributed Computing*, pages 12–20, 1989.

Fault-Tolerant Convergent Voting in Large Sparsely Connected Networks

R.M. Kieckhafer, and M.H. Azadmanesh

Department of Computer Science and Engineering
115 Ferguson Hall
University of Nebraska – Lincoln
Lincoln, NE 68588-0115

Abstract

In fault-tolerant distributed systems, different non-faulty processes may arrive at different values for a given system parameter. To resolve this disagreement, processes must exchange and vote upon their respective local values. During voting, faulty processes may attempt to inhibit agreement by acting in a malicious or "Byzantine" manner. Approximate Agreement defines a form of agreement in which the voted values obtained by the non-faulty processes need only agree to within a predefined tolerance. Approximate Agreement can be achieved by a sequence of convergent voting rounds, in which the range of values held by non-faulty processes is reduced in each round. Existing convergent voting algorithms assume complete connectivity between processes. Where the physical connectivity is incomplete, messages must be relayed between processors to simulate complete connectivity. For large, sparsely connected systems, the message traffic associated with message relaying could be prohibitive, making Approximate Agreement infeasible for such systems.

This paper presents a means of implementing convergent voting in large sparsely connected networks without the massive communication overhead incurred by the global relaying of messages. Simple expressions are presented for the convergence rates and robustness of a broad family of low-overhead *locally convergent* voting algorithms in the simultaneous presence of multiple fault modes. These expressions are employed to determine the robustness of local convergence in some commonly used partially connected networks. Issues affecting global convergence are also addressed, and the extension of the results to several related problems is discussed.

1 Introduction

As distributed computing systems become larger, there is a growing conflict between the increasing need for fault-tolerance and the computational overhead required by fault-tolerance mechanisms. Even in systems containing less than a dozen processing nodes, fault-tolerance overhead can consume a majority of the processing power [Pal85, Cze85]. In contrast, systems are now being built with hundreds or thousands of nodes. This paper presents a method of adapting one important fault-tolerance mechanism to very large systems while maintaining the overhead of a small system.

An important issue in distributed fault-tolerance is ensuring that all non-faulty processes agree on the values of critical data items despite active interference from faulty processes. This issue arises whenever non-faulty processes can legitimately form differing "opinions" regarding a specific value. They must then exchange and vote upon their local values to arrive at a single consensus value. If a faulty process is constrained to send the same erroneous value to all non-faulty processes, then simple majority voting is sufficient to provide immediate agreement. It is only necessary that the majority of the processes be non-faulty. Reaching agreement becomes significantly more difficult if a faulty process is permitted to send conflicting values to different non-faulty processes. A faulty process with this property has been called malicious, two-faced, Byzantine, or asymmetric.

The classic form of distributed agreement, *Byzantine Agreement*, requires that all non-faulty processes obtain identical voted values for any set of initial values. However, many applications do not require non-faulty processes to achieve exact agreement. Rather, they need only agree on a value to within a specified tolerance. This state, called *Approximate Agreement*, is useful in areas such as sensor data management and fault-tolerant clock synchronization [Kie88, Lam85, Lun84, Sch87, Tha89]. Given an arbitrarily small positive real value ϵ , Approximate Agreement is defined by two conditions [Dol83, Dol86]:

Agreement – The voting algorithms executed by all non-faulty processes eventually halt with voted values that are within ϵ of each other.

Validity – The voted value held by each non-faulty process is within the range of the initial values held by all non-faulty processes.

Most Approximate Agreement algorithms employ multiple rounds of message exchange interleaved with a convergent voting algorithm which guarantees that the range of values

held by the non-faulty processes is reduced in each round [Dol83, Dol86, Kie91, Lam85, Vas89]. This property, called single-step convergence, guarantees that the range of values will eventually be less than ϵ , given enough rounds.

Large distributed systems rely upon partially connected networks for interprocess communication [Fen81, Hwa84]. However, convergent voting algorithms have been derived only for systems which are completely connected. In systems with partial physical interconnections, the voting processes must relay messages such that the system connectivity is logically complete. If the total number of processes is large, the global exchange of local values can consume a great deal of time and communication resources. As a result, convergent voting algorithms are not practical in large sparsely connected systems.

Most analyses of convergent voting assume that *all* faults exhibit asymmetric or Byzantine behavior [Dol83, Dol86, Lam85]. In reality, asymmetry occurs only under complex and improbable conditions. Thus, if coincident faults occur, it is highly unlikely that all faults will be Byzantine in nature. Recently, the behavior of convergent voting algorithms has been analyzed in the simultaneous presence of three distinct modes of faults: asymmetric (Byzantine), symmetric, and benign (self-incriminating) [Kie91]. This analysis showed that convergent voting can be significantly more robust than predicted by the single-mode Byzantine fault model.

This paper presents a means for limiting the overhead of achieving Approximate Agreement in large sparsely connected systems. The general approach is to prohibit the relay of convergent voting messages. Thus, each processor performs convergent voting only with its immediate neighbors. The objectives are: (1) to present low-overhead convergent voting algorithms which function without message relay, (2) to analyze the convergence rates of these algorithms using a mixed-mode fault model, (3) to determine the theoretical bounds on their fault-tolerance as a function of the topology and connectivity of the network.

These results make Approximate Agreement feasible for very large distributed systems. They thus facilitate confident design and verification of distributed processes such as clock synchronization and redundant sensor management. In addition, the methodology employed is shown to be extendable to several related problems.

2 Background

Distributed systems have been partitioned into two distinct classes, *synchronous* and *asynchronous* systems [Dol83]. In a synchronous system there are finite bounds on the processing and communication delays of non-faulty processes. There is thus a point in time by which any process executing a convergent voting algorithm will have received all data from all non-faulty processes. Any data arriving after that time must have come from a faulty process. In an asynchronous system, no finite bounds on process operation exist, so that a process might have to wait "forever" to receive data from all non-faulty processes. It is thus impossible to differentiate between a slow non-faulty process and a "dead" faulty process.

This paper addresses synchronous systems only. The synchronous system model is most representative of real-time systems, and is applicable to both data voting and clock synchronization. In addition, synchronous systems require the participation of fewer processes than asynchronous systems [Dol83, Dol86]. This fact can be of great importance in sparsely connected systems, where communication is constrained. A similar approach can be applied to asynchronous systems to produce results analogous to those presented herein.

Most of the previous research in convergent voting assumed a completely connected network of processors. In systems with partial interconnection, messages must be relayed such that each non-faulty process receives a value from every other non-faulty process. Two approaches have been taken to reduce the overhead of global message exchange in partially connected systems. The first approach considers the system to be hierarchically composed of processor clusters [Shi87]. Within each cluster, all processors are completely connected. One processor in each cluster is also connected to one processor in another cluster such that the set of clusters is completely connected. It is then possible to set one tolerance on agreement within a cluster, and a looser tolerance on agreement between clusters. The second approach employs special purpose communication hardware to increase the efficiency of handling relayed messages [Ram90]. However, this approach does not reduce the overall complexity of the message traffic.

Several convergent voting algorithms have been derived, such as the Fault-Tolerant Mean [Dol83], the Fault-Tolerant Midpoint (Mean of Medial Extremes) [Dol83], the Interactive Convergence algorithm [Lam85], and Dolev's Optimal algorithm [Dol86]. Each algorithm required *ad-hoc* proofs of its fault-tolerance and convergence properties. Furthermore, these analyses were only valid under a single-mode Byzantine fault model.

In recent work, a large family of voting algorithms was defined, called *Mean-Subsequence-Reduced* (MSR) algorithms [Kie91]. The MSR family encompasses several of the previously known voting algorithms. Simple expressions have been derived for the fault-tolerance and convergence properties of *any* MSR algorithm under a mixed-mode fault-model. The work presented here extends this analysis to partially connected systems with no relay of voting messages. We begin with some necessary definitions and background.

2.1 Real-Valued Multisets

Approximate Agreement requires the manipulation of multisets of real values. A multiset is a collection of objects similar to a set. However, it differs from a set in that the elements of a multiset need not be distinct. For example, a set of real numbers contains no more than one occurrence of any given value, while a multiset of real numbers may contain multiple occurrences of the same value. The number of times a particular object (value) appears in a multiset is called the *Multiplicity* of that object. A finite multiset \mathbf{V} of real values may be represented as a mapping $\mathbf{V} : \mathbb{R} \rightarrow \mathbb{N}$. For each real value r , $V(r)$ is defined as the multiplicity of r in \mathbf{V} . The size of \mathbf{V} is $V = |\mathbf{V}| = \sum_{r \in \mathbb{R}} V(r)$.

An alternative representation for a multiset of real numbers is a monotonically increasing sequence of the real values of its elements, i.e. $\mathbf{V} = \langle v_1, \dots, v_V \rangle$ ordered such that: $v_i \leq v_{i+1} \forall i \in \{1, \dots, V-1\}$ [And63, Liu85]. Both representations of a multiset are equivalent, but for certain operations one form or the other is more convenient. To avoid confusion, we use upper-case symbols for multiplicities in the real-to-integer mapping form, e.g. $V(r)$. Similarly, we use angle-braces and lower-case symbols for elements in the sequence form, e.g. $\mathbf{V} = \langle v_1, \dots, v_V \rangle = \langle v_i \rangle \forall i \in \{1, \dots, V\}$.

Real-Valued Parameters – A multiset of real numbers has several useful real-valued parameters.

$\min(\mathbf{V}) = \min(r \in \mathbb{R} : V(r) > 0) = v_1$; the minimum value of the elements in \mathbf{V} .

$\max(\mathbf{V}) = \max(r \in \mathbb{R} : V(r) > 0) = v_V$; the maximum value of the elements in \mathbf{V} .

$\rho(\mathbf{V}) = [\min(\mathbf{V}), \max(\mathbf{V})] = [v_1, v_V]$; the real interval spanned by \mathbf{V} . $\rho(\mathbf{V})$ is called the *range* of \mathbf{V} .

$\delta(\mathbf{V}) = \max(\mathbf{V}) - \min(\mathbf{V}) = v_V - v_1$; the difference between the maximum and minimum values of \mathbf{V} . $\delta(\mathbf{V})$ is called the *diameter* of \mathbf{V} .

$\text{mean}(\mathbf{V})$ = The arithmetic mean of the real values of all elements of \mathbf{V} ;

$$\text{mean}(\mathbf{V}) = \frac{1}{V} \left(\sum_{r \in \mathcal{R}} V(r) \cdot r \right) = \frac{1}{V} \left(\sum_{i=1}^V v_i \right).$$

Multiset Relations – Two multisets \mathbf{U} and \mathbf{V} may be related by:

Union: Let $\mathbf{W} = \mathbf{V} \cup \mathbf{U}$. Then $W(r) = \max[V(r), U(r)] \forall r \in \mathcal{R}$.

Intersection: Let $\mathbf{W} = \mathbf{V} \cap \mathbf{U}$. Then $W(r) = \min[V(r), U(r)] \forall r \in \mathcal{R}$.

Sum: Let $\mathbf{W} = \mathbf{V} + \mathbf{U}$. Then $W(r) = V(r) + U(r) \forall r \in \mathcal{R}$.

Difference: Let $\mathbf{W} = \mathbf{V} - \mathbf{U}$. Then $\forall r \in \mathcal{R}$:

$$W(r) = \begin{cases} V(r) - U(r) & \text{if } V(r) > U(r) \\ 0 & \text{otherwise} \end{cases}$$

Subsequences – Given two sequences \mathbf{U} and \mathbf{V} , \mathbf{U} is a *subsequence* of \mathbf{V} if all elements of \mathbf{U} are selected from the elements of \mathbf{V} , and arranged in the same order as their relative order in \mathbf{V} . While a subsequence is also a submultiset, it has the important property that the *index* of an element in \mathbf{V} is the sole criterion for its inclusion in \mathbf{U} . Thus, the function which selects elements of \mathbf{V} to be included in \mathbf{U} is a mapping from the indices of \mathbf{U} to the indices of \mathbf{V} .

Formally, let $\mathbf{I}_V = \{1, \dots, V\}$ be the set of indices for multiset \mathbf{V} , and let $\mathbf{I}_U = \{1, \dots, U\}$ be the set of indices for multiset \mathbf{U} . Then, \mathbf{U} is a subsequence of \mathbf{V} if there is a fixed one-to-one (injective) mapping function $k: \mathbf{I}_U \rightarrow \mathbf{I}_V$ which preserves order. Thus, each index $i \in \{1, \dots, U\}$ corresponds to exactly one index $k(i) \in \{1, \dots, V\}$, where $k(i) \leq k(i+1)$. It follows that $u_i = v_{k(i)}$. Furthermore, since \mathbf{V} is a monotonically increasing sequence of real numbers, $u_i \leq u_{(i+1)} \forall i \in \{1, \dots, U-1\}$.

2.2 Multiple Mode Fault Model

In real-world systems, truly Byzantine behavior occurs only under highly improbable conditions. Accordingly, Meyer and Pradhan [Mey87] have partitioned the space of all possible faults into two distinct modes: *Benign* faults, defined as those which are self-incriminating,

or immediately self-evident to *all* non-faulty processes, and *Malicious* faults, defined as all faults which do not qualify as benign. Thambidurai and Park [Tha88] have further partitioned malicious faults into two sub-modes: *Symmetric* faults, whose behavior is perceived identically by all non-faulty processes, and *Asymmetric* faults, whose behavior may be perceived differently by different non-faulty processes.

The total number of faulty processes t in a system is given by $t = a + s + b$, where a is the number of asymmetric faults, s is the number of symmetric faults, and b is the number of benign faults. It has been shown that if all faults are treated as asymmetric, then convergence is possible only if $N \geq 3t + 1$ [Dol83]. This is the standard single-mode Byzantine fault model. Similarly, if all faults are treated as symmetric, then simple majority voting is sufficient, so that convergence can be guaranteed if $N \geq 2t + 1$. Finally, if all faults are benign, then only one non-faulty process is required, i.e. $N \geq t + 1$.

The results to be presented here are based on the simultaneous presence of all three fault modes. Using the mixed-mode fault model, N is determined as a function of a , s , and b , rather than t . This model is complete in that all possible fault modes are considered. However, it is not unrealistically conservative, as is the single-mode Byzantine fault model.

2.3 Convergence in Completely Connected Systems

Single-step convergence is formally defined in terms of the following.

- V_i = The multiset of values received in one round by arbitrary non-faulty process i .
- V = $|V_i|$. If less than V values are received, then an arbitrary default value is chosen for each missing value so that V is identical for all non-faulty processes.
- U_i = The submultiset of *correct* values in V_i , i.e. those values generated by non-faulty processes.
- $U_{i \cap j}$ = $U_i \cap U_j$, the multiset of correct values received identically by two processes i and j . With complete connectivity, $U_{i \cap j}$ is identical for all non-faulty processes. $U_{i \cap j}$ may thus be taken as the multiset of correct values received by *all* non-faulty processes.

Each non-faulty process i executes a convergent voting algorithm, producing voted value $F(V_i)$. A voting algorithm is convergent if both of the following conditions are true for every voting round:

[C1] For each non-faulty process i , the voted value is within the range of correct values, i.e. $F(\mathbf{V}_i) \in \rho(\mathbf{U}_{inj})$.

[C2] For each pair of non-faulty processes, i and j , the difference between their voted values is strictly less than the diameter of the multiset of correct values received, i.e. $|F(\mathbf{V}_i) - F(\mathbf{V}_j)| \leq C \delta(\mathbf{U}_{inj})$, where $0 \leq C < 1$.

Parameter C is the *Convergence Rate* of a voting algorithm, the primary measure of its effectiveness. The *Robustness* of a voting algorithm is the minimum number of processes N required to tolerate t faults.

2.4 MSR Voting Algorithms

In completely connected systems, convergence properties have been determined [Kie91] for an entire family of voting algorithms with the general form:

$$F(\mathbf{V}) = \text{mean}[Sel_{\sigma}(Red^{\tau}(\mathbf{V}))].$$

Function Red^{τ} , called the *Reduction* function, removes the τ largest and τ smallest elements from multiset \mathbf{V} . The function Sel_{σ} , called the *Selection* function, selects a submultiset of σ elements from the reduced multiset $Red^{\tau}(\mathbf{V})$. If Sel_{σ} produces a subsequence of $Red^{\tau}(\mathbf{V})$, then $F(\mathbf{V})$ is the *Mean* of a *Subsequence* of the *Reduced* multiset. Voting algorithms with this property are called *Mean-Subsequence-Reduced* (MSR) algorithms [Kie91].

Members of the MSR family differ from each other only in their definition of the selection function Sel_{σ} . Simple expressions have been found for the convergence rate and robustness of *any* MSR algorithm in a completely connected system. These results show that it is advantageous to discard recognized benign errors prior to voting provided that *all* processes do so. Thus, given a total of N processes containing b benign faults, $V = N - b$.

Robustness — For a completely connected synchronous system containing a asymmetric faults and s symmetric faults, it has been shown that an MSR voting algorithm can be convergent only if [Kie91]:

$$\tau \geq a + s \tag{2.1}$$

$$\sigma \begin{cases} \geq 1 & : a = 0 \\ \geq 2 & : a > 0 \end{cases} \tag{2.2}$$

$$V \geq 2\tau + \max(a + 1, \sigma). \quad (2.3)$$

Substituting the minimum allowable values for σ and τ from (2.1) and (2.2) into (2.3) shows that a convergent MSR voting algorithm may exist only if:

$$\tau \geq \tau_c \equiv a + s \quad (2.4)$$

$$V \geq V_c \equiv 3a + 2s + 1 \quad (2.5)$$

$$\text{hence : } N \geq N_c \equiv 3a + 2s + b + 1. \quad (2.6)$$

Convergence Rate — An important result of [Kie91] is the ease with which the convergence rate C can be determined for *any* MSR voting algorithm. To begin, we define the *Medial Multiset* $\mathbf{M} = \text{Red}^r(\mathbf{V}) = \langle m_1, \dots, m_M \rangle$, and the *Selected Multiset* $\mathbf{S} = \text{Sel}_\sigma(\mathbf{M}) = \langle s_1, \dots, s_\sigma \rangle$. By the definition of an MSR algorithm, \mathbf{S} is a subsequence of \mathbf{M} . Thus, if g is an index into \mathbf{S} , then for each integer $g \in \{1, \dots, \sigma\}$ there exists exactly one integer $k(g) \in \{1, \dots, M\}$ which guarantees that $s_g = m_{k(g)}$.

Given two indices $g, h \in \{1, \dots, \sigma\}$ where $g \leq h$, we define $\Delta k(g, h)$ as the number of elements in \mathbf{M} *spanned* by elements $\langle s_g, \dots, s_h \rangle$ in \mathbf{S} , i.e.

$$\Delta k(g, h) = [k(h) - k(g)]. \quad (2.7)$$

If $g = h$, then $\Delta k(g, h) = 0$. However, if $g < h$, then $\Delta k(g, h)$ is the number of elements of \mathbf{M} in the submultiset $\langle m_{k(g)+1}, \dots, m_{k(h)} \rangle$.

For a given number of asymmetric faults, a , it will be useful to know the minimum value of $(h - g)$ for which $\Delta k(g, h) \geq a$. Thus, for each $g \in \{1, \dots, \sigma\}$, we define the quantity Δ_g as follows:

IF: $\Delta k(g, \sigma) \geq a$,

THEN: Δ_g = the minimum value of $(h - g)$ such that $\Delta k(g, h) \geq a$,

ELSE: Δ_g does not exist for this value of g .

The ELSE clause is required because if $\Delta k(g, \sigma) < a$, then there is no $h \in \{g + 1, \dots, \sigma\}$ for which $\Delta k(g, h) \geq a$. Thus, if Δ_g exists, then $\Delta k(g, g + \Delta_g) \geq a$.

Finally, we define the parameter γ as:

$$\gamma = \max_{g \in \{1, \dots, \sigma\}} (\Delta_g). \quad (2.8)$$

Thus, given any $g \in \{1, \dots, \sigma - \gamma\}$, it is assured that $\Delta k(g, g + \gamma) \geq a$. In other words, the submultiset $\{s_g, \dots, s_{g+\gamma}\}$ is guaranteed to span a elements of \mathbf{M} , for all $g \in \{1, \dots, \sigma - \gamma\}$. It has been shown [Kie91] that the convergence rate of an MSR algorithm is:

$$C = \frac{\gamma}{\sigma}. \quad (2.9)$$

As a practical matter, obtaining the values of γ and σ is relatively simple, given constants M and a , and a specified selection function Sel_σ . Parameter σ is just the number of elements selected by $Sel_\sigma(\mathbf{M})$. To determine γ , one simply determines Δ_g for each $g \in \{1, \dots, \sigma\}$ by inspection and selects the maximum thereof. If there is no value of g for which Δ_g exists, then γ does not exist, and the algorithm is not convergent.

3 Local Convergence with Partial Connectivity

A partially connected system differs from a completely connected system in that a given process does not receive values from all non-faulty processes. Rather, it receives values only from a specific subset of processes. There are now two types of convergence to be considered: local convergence over a specified subgraph, and global convergence over the entire system graph.

This section presents theoretical bounds on the ability to achieve local convergence in partially connected systems. The results of [Kie91] are extended to include the quantitative impact of topological parameters such as degree. As before, simple expressions are obtained for the convergence rate and robustness of synchronous MSR algorithms under the mixed-mode fault model. For brevity, the results must be presented without proofs. The reader is referred to [Kie91a] for detailed proofs.

The following constraints are placed on the system:

1. The system topology is a non-hierarchical, regular, homogeneous, undirected graph of N processing nodes, each with degree d .
2. Messages received by a voting process may not be relayed to another process. Thus, the physical and logical connectivity are identical. Each voting process receives its own value as well as those of its immediate neighbors so that $V = d + 1$ for all non-faulty processes.

3. $N \gg V$ so that "wrap-around" effects can not assist the local convergence process in a given voting round.

The following sets and multisets describe the relationships between the values received by two nodes i and j in a partially connected system.

P_i = The set of processes whose values are receivable by process i (P_j is similarly defined for process j).

$P_{i \cap j} = P_i \cap P_j$, the set of processes whose values are receivable by both process, i and process j .

$P_{i \cup j} = P_i \cup P_j$, the set of processes whose values are receivable by process i , process j , or both.

$U_{i \cap j}$ = The multiset of all values generated by *non-faulty* processes in $P_{i \cap j}$.

$U_{i \cup j}$ = The multiset of all values generated by *non-faulty* processes in $P_{i \cup j}$.

In a completely connected system, $U_{i \cap j} \equiv U_{i \cup j}$. However, in a partially connected system, $U_{i \cap j} \subset U_{i \cup j}$. We therefore define two types of local convergence for partially connected systems.

Intersection Convergence: Given a voting algorithm $F(V)$, two processes i and j are *Intersection Convergent* if the following conditions are both true:

- [I1] $F(V_i) \in \rho(U_{i \cap j})$, and $F(V_j) \in \rho(U_{i \cap j})$,
- [I2] $|F(V_i) - F(V_j)| \leq C \delta(U_{i \cap j})$, where $0 \leq C < 1$.

Union Convergence: Given a voting algorithm $F(V)$, two processes i and j are *Union Convergent* if the following conditions are both true:

- [U1] $F(V_i) \in \rho(U_{i \cup j})$, and $F(V_j) \in \rho(U_{i \cup j})$,
- [U2] $|F(V_i) - F(V_j)| \leq C \delta(U_{i \cup j})$, where $0 \leq C < 1$.

A major difference between completely connected and partially connected systems is the strategy for handling benign faults. In a completely connected system, benign faults can be ignored because *all* processes can delete the benign errors from V and vote with a smaller sized multiset. However, in a partially connected system, *no* fault is self-evident

to *all* processes. Thus, ignoring self-evident faults would cause different processes to vote using different sized multisets so that V would not be identical for all processes. Thus, only symmetric and asymmetric faults are considered in this analysis, leaving $t = a + s$.

3.1 Intersection Convergence

The conditions necessary to ensure that two processes i and j are Intersection Convergent can be derived as a variant on the completely connected system previously described. We begin with the following definitions:

a = The number of asymmetrically faulty processes in $P_{i \cap j}$.

s = The number of symmetrically faulty processes in $P_{i \cap j}$.

χ = $|P_i| - |P_{i \cap j}| = |P_j| - |P_{i \cap j}|$, the number of processes whose values are receivable by either i or j , but *not* by both.

Each process, $x_i \in \{P_i \setminus P_{i \cap j}\}$ communicates with process i , but not with process j . Similarly, each process, $x_j \in \{P_j \setminus P_{i \cap j}\}$ communicates with process j , but not with process i . In the worst case, two processes x_i and x_j can send different values to processes i and j , respectively. Since x_i and x_j are not members of $P_{i \cap j}$, their values could be outside of $\rho(U_{i \cap j})$. Thus, each process pair (x_i, x_j) can have the same impact on V_i and V_j as a single asymmetrically faulty process in $P_{i \cap j}$. This effect can occur regardless of the fault status of x_i and x_j . There are χ such process pairs, which can behave like χ asymmetric faults. We thus define Δ'_g as the variant on Δ_g obtained by substituting $(a + \chi)$ for a .

IF: $\Delta k(g, \sigma) \geq (a + \chi)$,

THEN: Δ'_g = the minimum value of $(h - g)$ such that $\Delta k(g, h) \geq (a + \chi)$,

ELSE: Δ'_g does not exist for this value of g .

Parameter γ' is then the corresponding variant on γ , i.e.:

$$\gamma' = \max_{g \in \{1, \dots, \sigma\}} (\Delta'_g).$$

Thus, given any $g \in \{1, \dots, \sigma - \gamma'\}$, it is assured that $\Delta k(g, g + \gamma') \geq (a + \chi)$.

It can be shown that an MSR algorithm can be Intersection Convergent only if [Kie91a]:

$$\tau \geq [a + \chi] + s \quad (3.1)$$

$$\sigma \begin{cases} \geq 1 & : [a + \chi] = 0 \\ \geq 2 & : [a + \chi] > 0 \end{cases} \quad (3.2)$$

$$V \geq 2\tau + \max([a + \chi] + 1, \sigma). \quad (3.3)$$

Furthermore, the convergence rate of the algorithm is:

$$C = \frac{\gamma'}{\sigma}. \quad (3.4)$$

Using the minimum allowable values for σ and τ from (3.1) and (3.2), an Intersection Convergent MSR voting algorithm may exist only if:

$$\tau \geq \tau_I \equiv (a + s) + \chi \quad (3.5)$$

$$V \geq V_I \equiv 3a + 2s + 1 + (3\chi). \quad (3.6)$$

The minimum size of $\mathbf{P}_{i \cap j}$ can be derived from (3.6) by noting that $|\mathbf{P}_{i \cap j}| = V - \chi$. Thus, Intersection Convergence between processes i and j is possible only if:

$$|\mathbf{P}_{i \cap j}| \geq (3a + 2s + 1) + (2\chi). \quad (3.7)$$

3.2 Union Convergence

Union Convergence requires convergence within $\rho(\mathbf{U}_{i \cup j})$ rather than $\rho(\mathbf{U}_{i \cap j})$. Accordingly, it can be shown that Union Convergence is possible under less restrictive conditions than Intersection Convergence.

Two processes $x_i \in \{\mathbf{P}_i \setminus \mathbf{P}_{i \cap j}\}$ and $x_j \in \{\mathbf{P}_j \setminus \mathbf{P}_{i \cap j}\}$ can still send two different values to processes i and j respectively. However, if x_i and x_j are both non-faulty then both values are within $\rho(\mathbf{U}_{i \cup j})$. Thus non-faulty (x_i, x_j) pairs have less impact on Union Convergence than on Intersection Convergence.

We retain the previous definitions that a and s are the number of asymmetric and symmetric faults, respectively in $\mathbf{P}_{i \cap j}$. We then define:

f = The maximum number of faults in either $\{\mathbf{P}_i \setminus \mathbf{P}_{i \cap j}\}$ or $\{\mathbf{P}_j \setminus \mathbf{P}_{i \cap j}\}$, regardless of the fault modes exhibited.

It can be shown that an MSR algorithm may be Union Convergent only if [Kie91a]:

$$\tau \geq a + s + f \quad (3.8)$$

$$\sigma \begin{cases} \geq 1 & : [a + \chi] = 0 \\ \geq 2 & : [a + \chi] > 0 \end{cases} \quad (3.9)$$

$$V \geq 2\tau + \max([a + \chi] + 1, \sigma). \quad (3.10)$$

Applying the minimum values for σ and τ yields the bounds:

$$\tau \geq \tau_U \equiv (a + s) + f \quad (3.11)$$

$$V \geq V_U \equiv (3a + 2s + 1) + (\chi + 2f) \quad (3.12)$$

$$|\mathbf{P}_{inj}| \geq V - \chi = (3a + 2s + 1) + (2f). \quad (3.13)$$

If an MSR algorithm is Union Convergent, then the convergence rate is identical to that for Intersection Convergence, i.e. $C = \gamma'/\sigma$.

3.3 Summary

Table 1 summarizes the relevant parameters for convergence in completely connected systems and for Intersection Convergence or Union Convergence in partially connected systems. The listed bounds on τ and V are minima, beneath which convergence can not be guaranteed. These bounds are tight, because there exists an MSR algorithm which is convergent with the listed minimal parameters. That algorithm is the Fault-Tolerant Midpoint [Dol83], in which $Sel_\sigma(\mathbf{M})$ selects the two extrema of \mathbf{M} . Thus, $\mathbf{S} = \{m_1, m_M\}$ so that $\sigma = 2$, the minimum value allowed by (2.2), (3.2), and (3.9) for $a \geq 1$.

4 Network Examples

The results summarized in Table 1 show the general bounds on convergence in regular homogeneous network graphs. Applying these results to specific error scenarios in selected interconnection topologies illustrates the relative robustness of these networks for both types of local convergence.

4.1 Mesh Networks

Three common mesh networks are shown in Figure 1, with degrees $d = 4$, $d = 6$, and $d = 8$, respectively. Since each node also receives its own values, these degrees yield $V = 5$, $V = 7$,

Completely Connected	Partially Connected	
	Intersection	Union
$\tau_c = (a + s)$	$\tau_I = (a + s) + \chi$	$\tau_U = (a + s) + f$
$V_c = (3a + 2s + 1)$	$V_I = (3a + 2s + 1) + (3\chi)$	$V_U = (3a + 2s + 1) + (\chi + 2f)$
$ \mathbf{P}_{inj} \equiv N = V + b$	$ \mathbf{P}_{inj} \geq (3a + 2s + 1) + (2\chi)$	$ \mathbf{P}_{inj} \geq (3a + 2s + 1) + (2f)$
$\Delta k(g, g + \gamma) \geq a$	$\Delta k(g, g + \gamma') \geq a + \chi$	$\Delta k(g, g + \gamma') \geq a + \chi$
$C = \gamma/\sigma$	$C = \gamma'/\sigma$	$C = \gamma'/\sigma$

Table 1: Summary of Convergence Parameters

and $V = 9$, respectively. For each network, two nodes are selected and labelled i and j such that $|\mathbf{P}_{inj}|$ is maximized. In Figure 1, the nodes enclosed within a dashed box comprise \mathbf{P}_{inj} for that mesh.

Inspection of Figure 1 reveals that for the chosen (i, j) pair, the number of non-shared values received by each node is $\chi = 3$ in all three meshes. In the best case of a fault-free system, Table 1 shows that Intersection Convergence is possible only if $V \geq V_I = 3\chi + 1 = (3 \times 3) + 1 = 10$. Since $V_I > V$ for all three meshes, there exists no MSR voting algorithm which is Intersection Convergent for any of these systems.

In a fault free system, $V_U = \chi + 1 = 3 + 1 = 4$. Thus, all three meshes are Union Convergent in the fault-free case. Assuming a single-fault scenario, the worst case would be if that fault were asymmetric, in which case $V_U = 3a + \chi + 1 = 3 + 3 + 1 = 7$. Thus, in any single-fault scenario, both the Hexagonal and Octagonal meshes are Union Convergent. It can also be shown that the Octagonal mesh can tolerate a double fault as long as $a \leq 1$, while none of these networks can tolerate a double asymmetric fault or any triple fault.

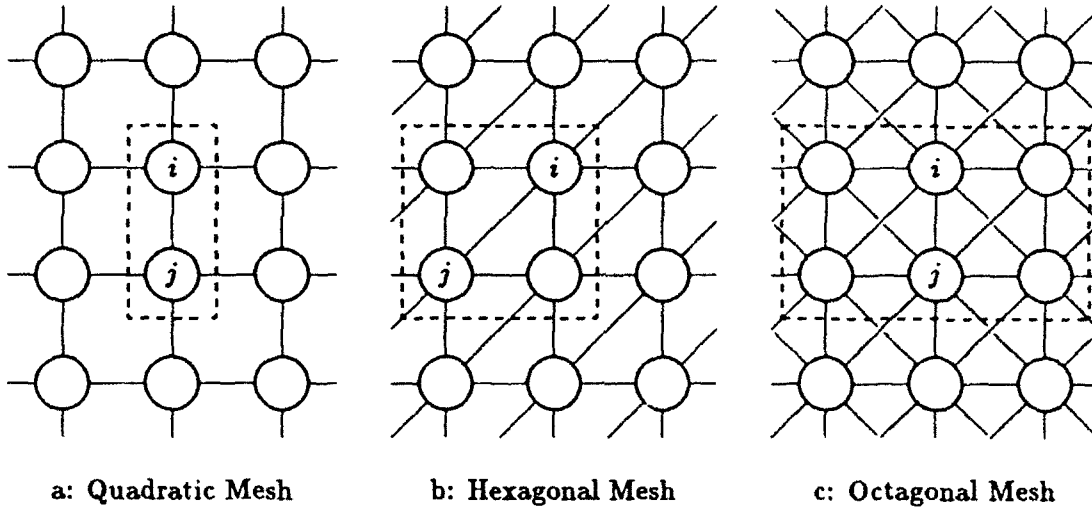


Figure 1: Common Mesh Networks

4.2 Hypercubes

In a hypercube, the degree $d = \log_2(N)$, so that $V = \log_2(N) + 1$. Each node is connected to all nodes whose binary address is at a Hamming distance of unity from its own address. Thus, for any two nodes i and j , $|\mathbf{P}_{ij}| \leq 2$. By definition, $\chi = V - |\mathbf{P}_{ij}| \geq V - 2 = \log_2(N) - 1$.

From table 1, the fault-free condition for Intersection-Convergence is $V_I = 3\chi + 1 = 3[\log_2(N) - 1] + 1$. Therefore, an Intersection Convergent MSR algorithm can exist for a hypercube only if:

$$\begin{aligned} V &\geq V_I, \\ V &\geq 3\chi + 1, \\ \log_2(N) + 1 &\geq 3[\log_2(N) - 1] + 1, \\ \log_2(N) &\leq \frac{3}{2}. \end{aligned}$$

Since $\log_2(N)$ must be an integer, the largest hypercube for which an Intersection-Convergent MSR algorithm could exist is defined by $\log_2(N) = 1$, or $N = 2$. This is the trivial system comprised of two nodes connected by a single link.

Performing a similar analysis for Union-Convergence yields:

$$\begin{aligned} V &\geq V_U, \\ V &\geq \chi + 1, \end{aligned}$$

$$\begin{aligned}\log_2(N) + 1 &\geq (\log_2(N) - 1) + 1, \\ \log_2(N) + 1 &\geq \log_2(N).\end{aligned}$$

Thus, any *fault-free* hypercube will be Union Convergent. However, any single fault within $P_{i,j}$ will add at least 2 to the right-hand side of this expression, making nodes i and j non-convergent.

5 Continuing Research

Table 1 shows the local convergence properties of MSR algorithms in regular homogeneous networks. These results and the methods used to obtain them are serving as the basis for further research on a number of related problems.

5.1 Global Convergence

For most system applications, the goal of convergent voting is to achieve Approximate Agreement on a global level, i.e. to reduce the range of values held by *all* non-faulty processes. While *local convergence* is a necessary pre-condition to global convergence, it is not by itself sufficient to guarantee global convergence. The existence and rate of global convergence also depend on the topology of the system, the distribution of initial values, and the distribution of faults throughout the system.

Single-step local convergence does not necessarily produce single-step global convergence. At the global level convergence may be *asymptotic* rather than immediate [Fek90]. There may be a delay of several rounds before the global diameter of correct values begins to decrease. Thus, a single-step convergence rate is an inappropriate performance metric for global convergence. Two metrics of interest are the maximum number of rounds required before the first reduction in global diameter, and the long-term average (or asymptotic) convergence rate. A sub-family of MSR algorithms has been identified which appears to minimize both of these metrics. Current efforts are directed at quantifying the performance of the global convergence process. Recent results suggest that global convergence can be guaranteed only if:

1. The fault-free system can achieve local Union Convergence between nearest neighbor nodes,

2. The number of errors received by any non-faulty node does not exceed the fault-tolerance limits set in Table 1 for Union Convergence with its nearest neighbors.

Examples have shown that if a single non-faulty node fails to meet condition 2 above, then the entire system will become non-convergent. These examples indicate that global convergence is extremely sensitive to the clustering of faults. General expressions describing the precise conditions required for global convergence to occur have not yet been derived.

5.2 Asynchronous Systems

In asynchronous systems, there are no bounds on the processing time or message delivery delay for non-faulty processes [Dol83]. Nonetheless, convergence is still possible. Using the single-mode byzantine fault model in completely connected systems, it has been shown that convergence is possible if $N \geq 5t + 1$. Current work indicates that for completely connected systems, the mixed-mode model yields $N \geq 5a + 4s + 1$. The methods used herein for synchronous systems can be extended to include mixed-mode partially connected asynchronous systems as well.

5.3 Non-Homogeneous Topologies

The results in Table 1 are based on the assumption that the network is regular and homogeneous. Thus, the degree and connectivity of all nodes are identical. Specifically, for any two voting processes, i and j , the size of the voting multiset V and the number of non-shared processes χ are identical.

If the network is not homogeneous, then one must deal with boundary conditions at the edges of the network. Worse still is an irregular network graph, in which case nodes i and j may have different-sized voting multisets. Thus, the interactions between voting algorithms in regions of differing connectivity are being investigated.

5.4 Omission Faults

In many systems, the most likely fault is an asymmetric omission fault caused by a faulty communication link. The known Approximate Agreement algorithms can not exploit this restriction on fault behavior. We are currently studying a variant of the MSR family of algorithms called Omission-MSR (OMSR). Previously, one OMSR algorithm has been

applied to clock synchronization in a system where asymmetric omission faults were known to be dominant [Tha88]. In that particular case the selected OMSR algorithm was more robust than would be possible for *any* MSR algorithm. Using methods similar to those employed herein, general expressions for the robustness and convergence rates of OMSR algorithms are being derived.

5.5 Partial Message Relay

At this point, two extreme approaches are known for achieving approximate agreement. The conventional approach employs *complete* message relay to emulate a completely connected network, while the approach used here employs *no* message relays. The complete relay approach offers faster convergence, while the no relay approach imposes lower message passing overhead. There is a continuum of *partial relay* policies lying between these two extremes. For example, nodes in a system may relay only those messages originated by an immediate neighbor. This approach can yield better robustness and convergence rates than the no relay approach, while still imposing much lower overhead than the complete relay approach.

6 Conclusions

This paper has presented a basis for limiting the overhead of achieving Approximate Agreement in large sparsely connected networks. These results make Approximate Agreement feasible in large Fault-Tolerant Real-Time systems. They thus facilitate the confident design and verification of distributed processes such as clock synchronization and redundant sensor management.

The general approach was to prohibit the relay of convergent voting messages so that each processor performs convergent voting only with its immediate neighbors. The main accomplishments were: (1) presentation of low-overhead convergent voting algorithms which function without message relay, (2) analysis of the convergence rates of these algorithms using the mixed-mode fault model, (3) determination of the theoretical bounds on fault-tolerance as a function of the topology and connectivity of the network.

The bounds for Intersection and Union Convergence shown in Table 1 apply to the entire MSR family of algorithms. Moreover, the required algorithmic and topological parameters are easy to determine, given a particular MSR algorithm and a particular topology. These

results provide the foundation for on-going research on a number of related problems.

References

- [And63] Anderson, K.W., and D.W. Hall, *Sets, Sequences, and Mappings: the Basic Concepts of Analysis*, New York, Wiley, 1963.
- [Cze85] Czech, E.W., et al., *Fault-Free Performance Validation of a Fault-Tolerant Multiprocessor: Baseline and Synthetic Workload Measurements*, Carnegie Mellon University, Dept. of Comput. Sci., Report CMU-CS-85-117, 1985.
- [Dol83] Dolev, D., et al. "Reaching Approximate Agreement in the Presence of Faults," *Proc. Third Symp. on Reliability in Distributed Software and Database Systems*, Oct 1983.
- [Dol86] Dolev, D., et al, "Reaching Approximate Agreement in the Presence of Faults," *JACM*, V. 33, No. 3, pp. 499-516, Jul 1986.
- [Fek90] Fekete, A.D., "Asymptotically Optimal Algorithms for Approximate Agreement", *Distributed Computing*, V. 4, pp. 9-29, 1990.
- [Fen81] Feng, T-Y, "A Survey of Interconnection Networks", *IEEE Computer*, V. 14, No. 12, pp. 12-27, Dec 1981.
- [Hwa84] Hwang, K., and F.A. Briggs, *Computer Architecture and Parallel processing*, McGraw-Hill, 1984.
- [Kie88] Kieckhafer, R.M., et al, "The MAFT Architecture for Distributed Fault-Tolerance", *IEEE Trans. Comput.*, V. C-37, No. 4, pp. 398-405, Apr, 1988.
- [Kie91] Kieckhafer, R.M., and M.H. Azadmanesh, *Reaching Approximate Agreement With Mixed Mode Faults*, University of Nebraska, Dept. of Comp. Sci. and Eng., Report Series No. 162, Nov, 1991.
- [Kie91a] Kieckhafer, R.M., and M.H. Azadmanesh, *Low Cost Approximate Agreement in Partially Connected Networks*, University of Nebraska, Dept. of Comp. Sci. and Eng., Report Series No. 164, Dec, 1991.
- [Lam85] Lamport, L., and P.M. Melliar-Smith, "Synchronizing Clocks in the Presence of Faults", *JACM*, Vol. 32, No. 1, pp. 52-78, Jan 1985.
- [Liu85] Liu, C.L., *Elements of Discrete Mathematics*, 2nd Ed., New York, McGraw-Hill, 1985.
- [Lun84] Lundelius, J., and N. Lynch, "A New Fault-Tolerant Algorithm for Clock Synchronization", *Third Symp. on Principles of Distributed Computing*, pp. 75-87, Aug 1984.

- [Mey87] Meyer, F.J., and D.K. Pradhan, "Consensus with Dual Failure Modes", *Proc. Seventeenth Fault-tolerant Computing Symposium*, pp. 48-54, Jul 1987.
- [Pal85] Palumbo, D.L., and R.W. Butler, *Measurement of SIFT Operating System Overhead*, NASA Tech. Memo 86322, 1985.
- [Ram90] Ramanathan, P., et al, "Hardware-Assisted Software Clock Synchronization for Homogeneous Distributed Systems", *IEEE Trans. Comput.*, V. C-39, No. 4, pp. 514-524, Apr 1990.
- [Sch87] Schneider, F.B., *Understanding Protocols for Byzantine Clock Synchronization*, Report No. 87-859, Dept of Computer Science, Cornell University, Aug 1987.
- [Shi87] Shin, K.G., and P. Ramanathan, "Clock Synchronization of a Large Multiprocessor System in the Presence of Malicious Faults", *IEEE Trans. Comput.*, V. C-36, No. 1, pp. 2-12, Jan 1987.
- [Tha88] Thambidurai, P.M., and Y.K. Park, "Interactive Consistency with Multiple Failure Modes", *Proc. Seventh Reliable Dist Systems Symp.*, Oct 1988.
- [Tha89] Thambidurai, P.M., et al., "Clock Synchronization in MAFT", *Proc. Nineteenth Fault-Tolerant Computing Symposium*, pp. 142-151, Jun 1989.
- [Vas89] Vasanthavada, N., and P., Thambidurai, "Design of Fault-Tolerant Clocks with Realistic Assumptions", *Proc. Eighteenth Fault-Tolerant Computing Symposium*, pp. 128-133, Jun 1989.

Applications and Extensions of the DRB Technology for Design of Real-Time Fault-Tolerant Distributed Computer Systems

K. H. (Kane) Kim
Department of Electrical & Computer Engineering
University of California
Irvine, CA 92717

Kane@Balboa.Eng.Uci.Edu

Abstract: The *distributed recovery block* (DRB) scheme initially formulated by the author in 1983 is a scheme devised for achieving *task-level real-time fault tolerance*. Under the scheme a fault arising in execution of a real-time task does not result in the failure of the timely delivery of an expected result to the receiver tasks since a redundant task is acting in a different node in a timely fashion. The DRB scheme can be used to obtain highly reliable real-time computing stations each capable of forward recovery from both hardware and software faults. Several demonstrations of the performance of the scheme in practical application contexts such as nuclear reactor control and defense command-control applications were conducted. In recent years efforts have been made to expand the application fields of the DRB scheme by extending the operational principles of and the implementation techniques for the scheme in several directions. The formulated extensions have been discussed piecemeal in different places. This paper is an attempt to take a comprehensive assessment of the extension efforts made and to present some newly formulated extensions and desirable directions for future extensions.

Area: Real-Time Fault-Tolerant Systems Design

1. Introduction

Many challenging real-time applications such as those encountered in defense and space exploration areas deal with "dirty" environments where electrical, electromagnetic, and mechanical disturbances cause relatively frequent failures of computer components. Therefore, *real-time fault tolerance* is a major requirement imposed on the computer systems used in such applications

unlike in commercial applications dealing with environments with low fault rates. Dramatic changes occurred in the past decade in the relative costs of hardware to software and those of VLSI processors to other peripheral and storage components eliminated many old problems that had long plagued the designers of complex real-time computer systems while they brought in new research issues. To mention a few examples:

(1) *Little incentives for multiprogramming and more for distributed processing:*

Because of the dramatically reduced hardware costs, complex software schemes such as multiprogramming techniques for heavy utilization of hardware are losing their appeals rapidly in many safety-critical real-time applications. In fact, it is now often much more cost-effective to design one-task-per-processor systems and such approaches make the temporal behavior analysis easy and they encourage high-level optimizations such as those aimed for faster guaranteed response. A node (or nodes) of a distributed computer system (DCS) dedicated to execution of an atomic real-time task is called a computing station in this paper.

(2) *Ease of using hardware redundancy:*

Again due to the relative low cost of hardware, it is now easier than before to use hardware redundancy for hardware fault tolerance, in particular, by use of active redundant hardware components. Techniques for hardware fault tolerance with the *forward recovery* effect such as the TMR (triple modular redundancy) scheme and the pair-of-comparing-pairs scheme have been extensively studied [And81, Car85, Toy87, Wil85].

(3) *Persistence of software reliability problems:*

Achieving an ultra-high reliability of real-time distributed software is still a serious challenge.

These technological and economic changes have also dictated concomitant changes in the most cost-effective unit of redundancy which also defines boundaries for fault containment and replacement for repair. Consequently, the focus in development of the fault-tolerant system design technology has been shifting from the techniques for utilizing *circuit-level* or *function-unit-level redundancy* through those for utilizing *processor-level redundancy* to those for *computing-station-level redundancy*.

The *distributed recovery block* (DRB) scheme initially formulated by the author in [Kim84] is a scheme for exploiting redundancy at the computing station level to achieve *task-level fault tolerance*. Under the scheme a fault arising in execution of a real-time task does not result in the failure of the timely delivery of an expected result to the receiver tasks since a redundant task is acting in a timely fashion. The DRB scheme can be used to obtain highly reliable computing stations each capable of forward recovery from both hardware and software faults. Several demonstrations of the performance of the scheme in practical application contexts were conducted [Kim88, Kim89a, Hec89, Hec91, Kim91a].

In recent years efforts have been made to expand the application fields of the DRB scheme by extending the operational principles and the implementation techniques for the scheme in several directions. The formulated extensions have been discussed piecemeal in different places. This paper is an attempt to take a comprehensive assessment of the extension efforts made and to present some newly formulated extensions and desirable directions for future extensions.

The paper starts with an overview of the basic DRB scheme and its application fields in Section 2. In Section 3, five major extensions of the DRB scheme are discussed and remaining issues related to full development of the extensions are discussed. Thereafter, a simplified application of the DRB scheme to *highly parallel multi-computer networks* (HPM's) in order to realize fault-tolerant execution of a large number of real-time tasks is discussed. The final section concludes with a discussion on four desirable directions for future extension of the DRB technology.

2. Basic DRB Scheme

2.1 Basic principles

The distributed recovery block (DRB) scheme is based on a combination of both distributed concurrent processing and recovery block structuring concepts to achieve fast forward error recovery and to treat both hardware and software faults in a uniform manner with minimal execution overhead [Kim84, Kim89a]. It is an active redundancy scheme where multiple processors concurrently execute multiple versions of a software component and then the same reasonableness check routine. The reasonableness check routine in each processor, together with a watch-dog timer, checks reasonableness of the computational results of the version executed as well as the timeliness of the execution.

Recovery block consists of one or more routines, called try blocks here, designed to compute the same or similar results, and an acceptance test which is an expression of the criterion used for accepting the results of try blocks [Hor74, Ran75]. A *try* (i.e., execution of a try block) is thus always followed by an acceptance test execution. If an error is detected during a try or as a result of an acceptance test execution, then a rollback-and-retry with another try block follows. Therefore, it is an enclosure of some recoverable activities of a process and facilitates *backward recovery* and *software fault tolerance*.

Under the DRB scheme, a recovery block is replicated into multiple nodes forming a *DRB computing station* for parallel processing. In most cases a recovery block containing just two try blocks, i.e., the primary and the alternate, is designed and then assigned to two different nodes called the primary and shadow nodes as depicted in Figure 1. The roles of the two try blocks are assigned differently in the two nodes. Primary node X uses try block A as the first try block initially, whereas shadow node Y uses try block B as the initial first try block. Therefore, until a fault is detected, both nodes receive the same input data, process the data by use of two different try blocks (i.e., block A on node X and block B on node Y), and check the results by use of the acceptance test. Both nodes perform all these tasks concurrently. The time acceptance

test (i.e., the time-out mechanism) is used to ensure timely behavior of both nodes.

In a fault-free situation, both nodes will pass the acceptance test with the results computed with their first try blocks. In such a case, the primary node notifies the shadow of its success in the acceptance test. Thereafter, only the primary node sends its output to the successor computing station(s). Both nodes then proceed to the next task cycle. However, if the primary node fails and the shadow node passes its own acceptance test, the shadow node assumes the role of the primary, i.e., the nodes exchange their roles. To be more specific, upon its failure in passing the acceptance test the primary node attempts to inform the shadow node. The shadow node will take over the role of the primary as soon as it receives the notice. If the primary node crashes completely, the shadow node will recognize the failure of the primary upon expiration of the preset time limit. It will then become the new primary. These interactions between two nodes are done asynchronously. On the other hand, if the shadow node fails first, the primary node need not be disturbed. In both cases, the failed node attempts to become an operational shadow node; it attempts to roll back and retry with its second try block to bring its application computation state including local database up-to-date. This attempt does not disturb the primary node.

This approach has two useful characteristics:

- a) Recovery can be accomplished in the same manner regardless of whether a node fails due to hardware faults or software faults;
- b) The recovery time is minimal since maximum concurrency is exploited between the primary and the shadow nodes.

In recent years basic techniques for implementation of the DRB scheme were established and several demonstrations of the performance of the scheme in practical application contexts were conducted. For example, several experiments were conducted at the author's location and they involved application of the DRB scheme to adjacent computing stations in real-time parallel processing multi-computer testbeds [Kim88, Kim89a, Kim91]. Figure 2 depicts one such testbed. The hardware base of the testbed contains several global shared memory modules which facilitate inter-node communication. Each data queue serving as a link among logically

adjacent nodes is housed in one of the global shared memory modules. The distributed real-time application software in this testbed was designed to perform continuous control of the optical sensors and the vehicle guidance and navigation subsystem in order to track a rapidly moving target. The results demonstrated the fast forward recovery capability of the DRB scheme as well as the effectiveness of the implementation approaches formulated.

Another major validation was conducted by a small company located in Los Angeles, SoHaR, Inc. They extended the DRB scheme for use in real-time local area PC networks for nuclear reactor control applications and produced a product prototype [Hec89, Hec91]. Figure 3 depicts a high level view of such networks.

2.2 DRB stations in HPM'S vs. DRB stations in LAN'S

Since the DRB scheme is a technique for realizing a "hardened" real-time computing station and since both real-time computer systems based on highly parallel multi-computer networks (HPM's) and those based on local area networks (LAN's) can also be structured in the natural form of interconnections of real-time computing stations, the application fields of the DRB scheme cover both HPM based applications and LAN based applications. On the other hand, the differences in interconnection structures and mechanisms between the HPM's and the LAN's can have impacts on the approaches for implementation of DRB computing stations.

In LAN based systems, the inter-node communication costs are greater and the costs of providing redundant communication paths are greater. Therefore, the overhead of synchronizing the primary and shadow nodes at the beginning of each task cycle as well as the overhead for status exchange is much greater in LAN based systems than in HPM based systems. Also, in some HPM's, nodes may be connected via shared memory modules as is the case in Figure 2. In such HPM based systems, data queues hosted on shared memory modules serve as communication media between DRB stations as well as between partner nodes belonging to the same DRB stations. Data queue management should be done such that not only the partner nodes stay

synchronized to an acceptable degree but also they preserve *input data consistency* in the sense that they always pick the same data item or copies of the same data item at each task cycle [Kim91].

3. Major Established Extensions of the DRB Scheme

Five major established extensions of the DRB scheme are assessed in this section.

3.1 DRB computing station based on comparing processor-pairs

The basic DRB scheme relies on the logic acceptance test and the time acceptance test for error detection. For faster detection of hardware faults, the DRB scheme can be extended to incorporate various established mechanisms. A hardware fault detection scheme that has been solidly established and has met continuously growing acceptance due to the improved hardware economy, is the *comparing processor-pair* scheme [Toy78, Wil85]. This scheme has been incorporated into commercial industry products widely in use such as AT&T ESS and Stratus computers. An extension of the DRB scheme under which each of the nodes (primary and shadow) in a DRB station is implemented in the form of a comparing processor-pair is depicted in Figure 4. Such an DRB computing station should exhibit much shorter *detection latency* for most hardware faults than the ordinary DRB station does. Therefore, in the DRB station shown in Figure 4, only some rare types of hardware faults and software faults will escape the guards set by the comparing processor-pair mechanism and will have to be detected by the acceptance test with concomitant larger detection latencies.

One thing to note here is that faster detection of hardware faults occurring in a node within a DRB station will be useful only to the extent that the failed node can initiate sooner its attempt to become a shadow node. However, another important attractive feature of the comparing processor-pair mechanism is its high coverage in detecting hardware faults.

3.2 Multiple recovery blocks in a DRB computing station

For reasons such as efficient node utilization or special characteristics of the applications, often multiple tasks may be designed to reside on the same node in spite of the fact that the single-task-per-node approach is becoming justified with increasing ease. This means that multiple recovery blocks may reside in a DRB station [Kim91]. Actually, the following three cases are conceivable.

(1) Multi-procedure DRB station: Each of multiple recovery blocks in the same DRB station is provided to process data items from a different source (predecessor computing station) or to process a different type of data items. The motivation for structuring this type of DRB stations is the node economy. The application software of a multi-procedure DRB station thus takes the form of a "case" statement enclosing multiple recovery blocks as depicted in Figure 5(a). A multi-procedure DRB station is depicted in Figure 5(b).

(2) Multi-phase DRB station: This case arises where the mission life of a task running on a computing station consists of multiple phases and different phases require substantially different processing algorithms. The operations for each phase can be naturally designed into a separate recovery block. Although it is possible to form a separate DRB station around each recovery block, it is a wasteful approach since there is no parallelism among such DRB stations. Therefore, a multi-phase DRB station can be viewed as one running a single task structured in the form of a "case" statement enclosing multiple recovery blocks. Figure 5(a) and 5(b) are thus applicable to a multi-phase DRB station also.

(3) DRB station with serially bonded recovery blocks: This DRB station contains multiple recovery blocks connected in a series form as shown in Figure 6. Such recovery blocks are called serially bonded recovery blocks. This case naturally arises where a task is required to deliver its processing results at several different stages, possibly to different destinations. This DRB station structuring can be motivated not only for node economy but also for improved data turnaround time. To be more specific, if two recovery blocks closely related in the form of a procedure-consumer relation are assigned to two

separate DRB stations, then message communication between recovery blocks involves inter-node communication. In LAN-based systems, such inter-node communication delay is significantly larger than the intra-node communication delay which would be incurred when both recovery blocks are assigned to one DRB station. Therefore, the single DRB station approach may lead to a shorter data turnaround time from the input action of the first recovery block to the output action marking the end of the second recovery block execution. On the other hand, the arrival rate of input data for a DRB station with serially bonded recovery blocks must be constrained such that the average inter-arrival time is substantially larger than the execution time for all the serially bonded recovery blocks combined together.

The above three types of extended structuring options are believed to widen the application fields of the DRB scheme considerably. It is also important to note that various combinations of the three options are feasible although detailed implementation issues and cost-performance issues need to be studied in the future.

3.3 N try blocks in a DRB computing station

In some highly safety-critical applications, the system designer may design more than two try blocks into a recovery block for the sake of increased reliability and comfort. Although several approaches to structuring a DRB station that uses three try blocks are conceivable, one of the most natural approaches is to treat the third node as a shadow node for the team of the first two nodes. Such a station is depicted in Figure 7.

Node Z in the figure will normally use try block C as its primary try block and deliver its results only when both X and Y fail to produce acceptable results in time. Nodes X and Y behave like a single functional node with respect to interfacing with their shadow node Z. They must share responsibilities for providing their status information to node Z at various points as well as responsibilities for understanding the status of node Z. For example, the type of an input data item picked, the acceptance test result (an indication enabling node Z to determine if any one of the two nodes X and Y has passed its acceptance test), the success of delivering the result by node X or Y to the successor stations,

etc., are the information that needs to be provided to node Z.

If node X or Y crashes, then it can be replaced by node Z and thus the station can start functioning as an ordinary two-node DRB station. Similarly, crash of node Z will result in the station functioning as an ordinary two-node DRB station. If both X and Y fail at their acceptance tests but are alive, then node Z becomes the new primary node and one of the two failed nodes X and Y should become the new secondary node (a shadow for node Z) and the other the third node (a shadow for the team of Z and the other node). The time-out value used by the third node Z waiting for status information from the team of X and Y can be somewhat larger than that used by Y monitoring the primary node X.

An important advantage of the approach depicted in Figure 7 is the recursive nature of the approach. Therefore, in an n-node DRB station, the n-th node functions as a shadow for the team of the first n-1 nodes. A natural consequence of this recursive organization is the modest increase in the implementation complexity as the number of nodes used in a DRB station increases.

3.4 Adaptive DRB computing station

In some applications, environmental conditions that affect fault tolerance requirements imposed on computer systems change dynamically. As significant changes in environmental conditions or in internal computing resource conditions occur, the set of fault tolerance mechanisms that are effective also changes.

An interesting concept for extending the DRB scheme is the dynamic switching between the recovery block scheme and the DRB scheme in response to changes in resource constraints [Kim92a, Arm91]. The DRB scheme requires more processing nodes than the recovery block scheme but facilitates forward recovery. Therefore, the recovery block scheme can be used in the *soft-real-time mode* while the DRB scheme can be used in the *hard-real-time mode*. However, if the number of processing nodes available falls below a certain threshold while the system is operating in the hard-real-time mode, then the system may switch from the

DRB scheme to the recovery block scheme for execution of selected tasks. The adaptive version of the DRB scheme which can transit among several modes of operation with concomitant changes in resource consumption and recovery performance, is called here the adaptive DRB scheme.

Let us now consider an approach to facilitating the switching between the recovery block scheme and the DRB scheme. Figure 8 depicts such an approach. As shown, a node executes status-1, status-2, check-1*, and status-3 actions only when it is operating as the primary node of a DRB station. Its mode of operation can be determined by checking if the valid ID of the shadow-partner node is in the relevant data structure. Similarly, a node can be either in the mode of functioning as the shadow node of a DRB station supporting the primary-partner node or in the simplex mode of executing another task (or recovery block) independently. When the node is ordered to be in the latter mode by the system resource allocator, the node should also be given an instruction as to which task (or recovery block) to execute.

Therefore, when the system resource allocator converts a node executing a recovery block in the simplex mode into the primary node of a DRB station, the following steps are involved. The system resource allocator first designates a node to become the shadow node of the DRB station. The allocator activates the shadow node first by informing the node of the ID of its primary-partner node. This activation may or may not involve an abortion of an on-going task execution. Thereafter, the allocator instructs the primary node to supply the necessary computation state to the shadow-partner and then start cooperative redundant execution.

3.5 Integration of the DRB scheme and network configuration management schemes

In order to shorten fault detection latency and further enhance the survival period of the DCS, the DRB scheme must be integrated with techniques for network configuration management (NCM). The NCM function generally involves detecting crashed nodes, whether they were in busy (non-idling) states before the crashes or not, and reincorporating

repaired nodes into the operating network configuration.

The integration of the DRB scheme and a practical centralized NCM scheme has been developed by SoHaR, Inc. [Hec89, Hec91]. The centralized NCM approach has an advantage of its simplicity but can become a single point of failure.

Several decentralized approaches to NCM have also been studied in recent years [Cri88, Jen89, Kop89, Kim92b]. Integrations of the DRB scheme with such decentralized NCM schemes have yet to be accomplished.

Once a node in a DRB station is functionally or physically amputated off for repair, then the system resource allocator attempts to find a replacement node. Such restructuring must be implemented in a architecture-dependent manner since efficient synchronization and efficient status exchange between the partner nodes in a DRB station are always desirable. Moreover, the issue of non-disruptive rejoin, i.e., incorporating a new node into a DRB station and conditioning it into an active shadow node without disturbing the primary node much is a non-trivial one. Therefore, implementation of a repairable DRB station is a subject awaiting much further study.

4. A Simplified Application of the DRB scheme to HPM's

The applicability of the DRB scheme to the HPM's for fault-tolerant execution of real-time tasks was already mentioned in Section 2. The DRB scheme can be viewed as a software-implemented approach for achieving fault tolerance in HPM's without requiring special hardware mechanisms. An important point here is that in applying the DRB scheme to an HPM, the scheme need not be utilized in its full generality. To be more specific, if the system developer is not concerned with possible software faults, then alternate try blocks are not necessary. Only one algorithm needs to be designed for each task.

Moreover, it is not a requirement to design a task-specific acceptance test. A common acceptance test designed to perform spot checks on a few selected areas of the machine hardware or integrity checks for various data structures can

be executed at the end of each task to decide whether to trust the task result as an acceptable one or not. In such a case where a task-independent common acceptance test is used and no alternate try blocks are used, forming a DRB station dedicated to fault-tolerant execution of a task becomes a mechanical process which does not burden the application software designer in any way. This approach can thus be viewed as a concrete approach to mechanical replicated execution of real-time tasks in HPM's. The cooperation between the partner nodes follows the same protocol discussed in Section 2.

An important issue in designing a HPM based real-time system is that of mapping tasks to the nodes of the HPM. When the aforementioned mechanized formation of DRB stations is used, task mapping involves assignment of each task to two nodes in reasonable close proximity [Kim90]. The distance between partner nodes must be short because it impacts directly the efficiency of synchronization and that of status exchange between the partners and thus impacts the data turnaround time also.

5. Summary and Future Extension

The DRB scheme is a basic technology for realizing a real-time fault-tolerant computing station which is a component of a real-time DCS and as such can receive input data from and send computation results to other computing stations in the same DCS. It has been evolved into a broadly applicable technology in the past nine years and has been demonstrated via several testbeds and one product prototype. The extended DRB schemes reviewed in Section 3 have significantly broader application fields than the basic DRB scheme does. However, it is fair to say that the DRB scheme is by and large a technique specialized for safety-critical real-time applications and not yet a fully matured technology. The following directions are considered to be among the most important for bringing the DRB technology to a more mature (more widely and easily practicable) form.

(1) Integration of the DRB scheme and decentralized NCM schemes

The interesting development by SoHaR, Inc on an integration of a centralized NCM scheme

and the DRB scheme was mentioned earlier. There are a broad range of real-time LAN applications where decentralized NCM is desirable. Therefore, there are needs for establishing efficient decentralized approaches to NCM and integrating them with the DRB scheme.

(2) Highly adaptive DRB stations

Further extension of the integration task in (1) is to fully establish the technique for structuring adaptive DRB stations discussed in Section 3.4. In a highly adaptive DRB station, at least three different modes of operation are conceivable: sequential backward recovery mode (recovery block scheme), concurrent processing forward recovery mode (DRB scheme), and sequential forward recovery mode in which a specially designed application-level recovery routine is invoked without automatic, fully application-transparent rollback upon failure of the primary routine to produce an acceptable result. More importantly, the criteria used for making decisions for switching among the three modes must be related to resource conditions among others. They need to be established in concrete forms and validated in future research.

(3) Integration with the object-based structuring concept

Object-based structuring approaches are meeting increasing acceptance from system designers for reasons such as modularity, etc. This is the case for both soft-real-time systems and hard-real-time systems [Kop90]. Adaptation of the DRB scheme to an object-based approach for structuring of real-time tasks is an important subject for future study.

(4) Distributed conversation (DCONV) scheme

The DRB scheme is applicable to non-interacting segments (i.e., atomic tasks) of application processes. To put it another way, it is a scheme to prevent a fault from crossing the boundaries between real-time processes as much as possible. For protecting against faults leaking through the guards established by the DRB scheme, supplementary schemes are needed. A promising case of a supplementary scheme is the distributed conversation (DCONV) scheme [Kim89b] which is essentially a combination of the conversation structuring scheme [Ran75] and the approach of concurrent execution of

redundant software components which was exploited in the DRB scheme. In a sense, the DCONV scheme can be viewed as an approach to hardening a group of interacting computing stations. The scheme is capable of achieving forward recovery when a part or all of a group of computing stations fail. The research in this scheme is however in its early stage.

Acknowledgement: This work was supported in part by the University of California MICRO program under Grant No 91-075.

References

- [And81] Anderson, T. and Lee, P.A., 'Fault Tolerance: Principles and Practice', Prentice-Hall Int'l, Inc., London, 1981.
- [Arm91] Armstrong, L.T. and Lawrence, T.F., "Adaptive Fault Tolerance", Proc. 1991 Systems Design Synthesis Technology Workshop, Sept. 1991.
- [Car85] Carter, W.C., "Hardware Fault Tolerance", Chapter 2 in Anderson, T., ed., 'Resilient Computing Systems', Vol. 1, Wiley-Interscience, 1985, pp.11-63.
- [Cri88] Cristian, F., "Agreeing on Who is Present and Who is Absent in a Synchronous Distributed System", Proc. IEEE Computer Society's 18th Int. Symp. of Fault-Tolerant Computing, Tokyo, Japan, June 1988, pp. 206-211.
- [Hec89] Hecht, M., Agron, J., and Hochhauser, S., "A Distributed Fault Tolerant Architecture for Nuclear Reactor Control and Safety Functions", Proc. IEEE Computer Society's 1989 Real-Time Systems Symp., Dec. 1989, pp.214-221.
- [Hec91] Hecht, M. et al., "A Distributed Fault Tolerant Architecture for Nuclear Reactor and Other Critical Process Control Applications", Proc. IEEE Computer Society's 21st Int'l Symp. on Fault-Tolerant Computing, June 1991, Montreal, pp.462-469.
- [Hor74] Horning, J.J., Lauer, H.C., Melliar-Smith, P.M., and Randell, B., "A Program Structure for Error Detection and Recovery," Lecture Notes in Computer Science, Vol.16, Springer-Verlag, New York, NY, 1974, pp. 171-187.
- [Jen89] Jensen, D. and Northcutt, J.D., "Alpha: An Open Operating System for Mission-Critical Real-Time Distributed Systems - An Overview", Proc. 1989 Workshop on Operating Systems for Mission-Critical Computing, ACM Press, 1991.
- [Kim84] Kim, K.H., "Distributed Execution of Recovery Blocks: an Approach to Uniform Treatment of Hardware and Software Faults", Proc. 4th Int'l Conf. on Distributed Computing System, May 1984, pp.526-532.
- [Kim88] Kim, K.H. and Yoon, J.C., "Approaches to Implementation of a Repairable Distributed Recovery Block Scheme", Proc. 18th Int'l Symp. on Fault-Tolerant Computing (FTCS-18), pp.50-55.
- [Kim89a] Kim, K.H. and Welch, H.O., "Distributed Execution of Recovery Blocks: An Approach to Uniform Treatment of Hardware and Software Faults in Real-Time Applications", IEEE Trans. on Computers, May 1989, pp.626-636.
- [Kim89b] Kim, K.H., "Approaches to System-Level Fault Tolerance in Distributed Real-Time Computer Systems", the Proc. 4th Int'l Conf. on Fault-Tolerant Computing Systems, Baden-Baden, W. Germany, Sept. 1989, pp.268-281 (published in the Lecture Notes series by Springer-Verlag) (Invited paper).
- [Kim90] Kim, K.H. and Kavianpour, A., "A Distributed Recovery Block Approach to Fault-Tolerant Execution of Application Tasks in Hypercubes", Tech. Rept. UCI-ECE-90-4, Dept. of Electrical & Computer Engineering, UCI, April 1990. To appear in IEEE Trans. on Parallel and Distributed Systems.
- [Kim91] Kim, K.H. and Min, B.J., "Approaches to Implementation of Multiple DRB Stations in Tightly Coupled Computer Networks and an Experimental Validation", Proc. IEEE Computer Society's 15th Int'l Computer Software and Applications Conf. (COMPSAC 91), Sept. 1991, Tokyo, pp.550-557.
- [Kim92a] Kim, K.H. and Lawrence, T.F., "Adaptive Fault-Tolerance in Complex Real-Time Distributed Computer System Applications", Computer Communications, Vol.15, No.4, May 1992, pp.243-251. (A shorter version entitled "Adaptive Fault Tolerance: Issues and Approaches" appeared in Proc. 2nd IEEE Computer Society's Workshop on Future Trends of Distributed Computing Systems, Cairo, Egypt, Sept. 1990, pp.38-46.)

[Kim92b] Kim, K.H., Kopetz, H., Mori, K., Shokri, E.H., and Gruensteidl, G., "An Efficient Decentralized Approach to Processor-Group Membership Maintenance in Real-Time LAN Systems: The PRHB/ED Scheme", To appear in Proc. 11th IEEE Computer Society's Symp. on Reliable Distributed Systems, Houston, Oct. 1992.

[Kop89] Kopetz, H., G. Gruensteidl and J. Reisinger, "Fault-tolerant membership service in a synchronous distributed real-time system", Proceedings of the International Working Conference on Dependable Computing for Critical Applications, Santa Barbara, CA, August 1989, pp.167-174.

[Kop90] Kopetz, H. and Kim, K.H., "Temporal

Uncertainties in Interactions among Real-Time Objects", Proc. IEEE Computer Society's 9th Symp. on Reliable Distributed Systems, Huntsville, AL, Oct. 1990, pp.165-174.

[Ran75] Randell, B., "System Structure for Software Fault Tolerance", IEEE Transactions on Software Engineering, June 1975, pp.220-232.

[Toy78] Toy, W.N., "Fault-Tolerant Design of Local ESS Processors", Proceedings of the IEEE, Vol.66, No.10, Oct. 1978, pp. 1126-1145.

[Toy87] Toy, W.N., "Fault-Tolerant Computing", A chapter in Advances in Computers, Vol. 26, Academic Press, 1987, pp.201-279.

[Wil85] Wilson, D., "The STRATUS computer system", Chapter 12 in T. Anderson ed., 'Resilient Computing Systems Volume I', John Wiley & Sons, 1985.

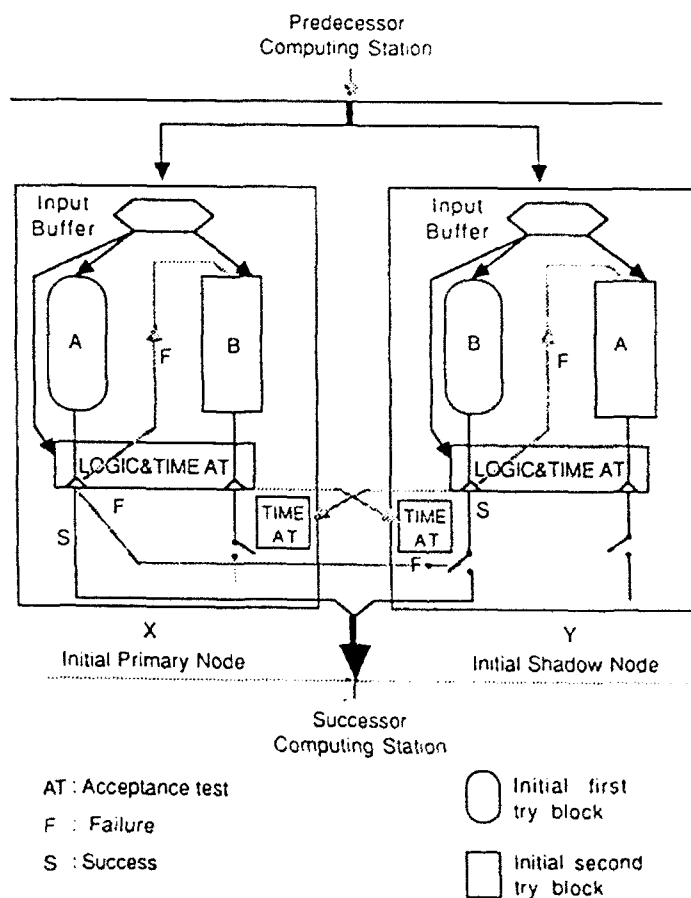


Figure 1. The basic DRB computing station
(Adapted from [Kim89a])

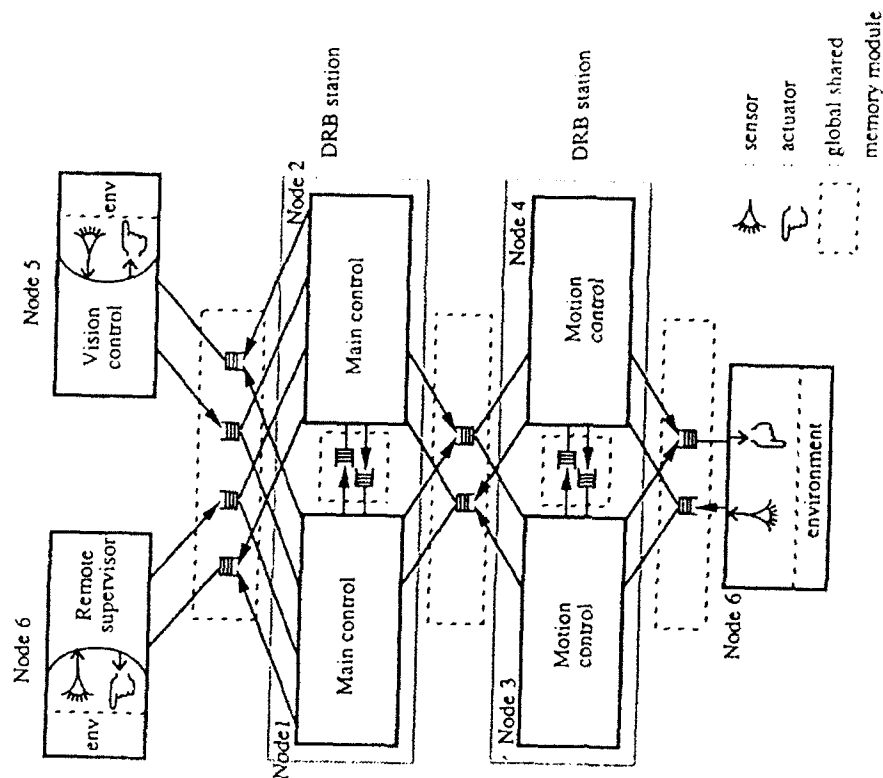


Figure 2. A DRB-based fault-tolerant configuration of a vehicle control system tested (Adapted from [Kim91])

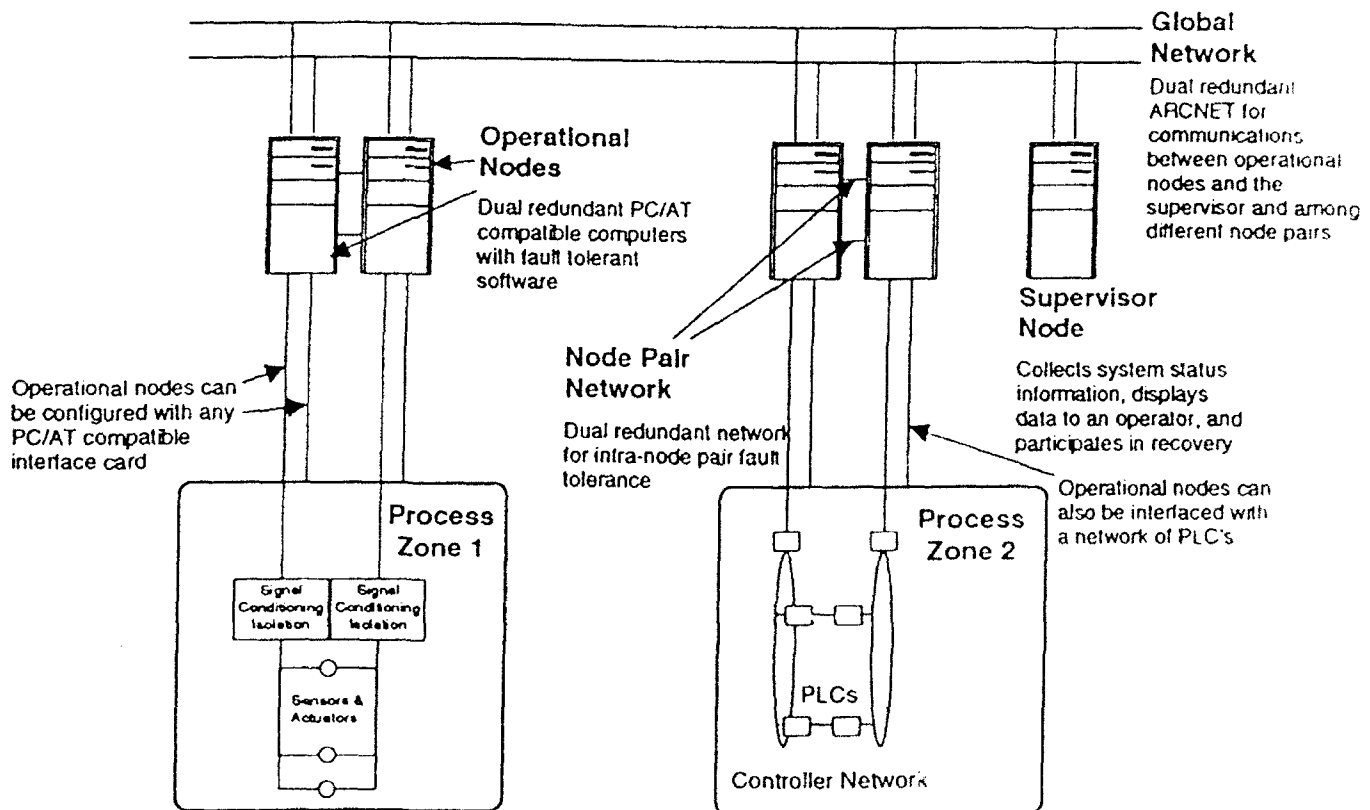


Figure 3. The DRB-based fault-tolerant LAN architecture developed by SoHaR, Inc.

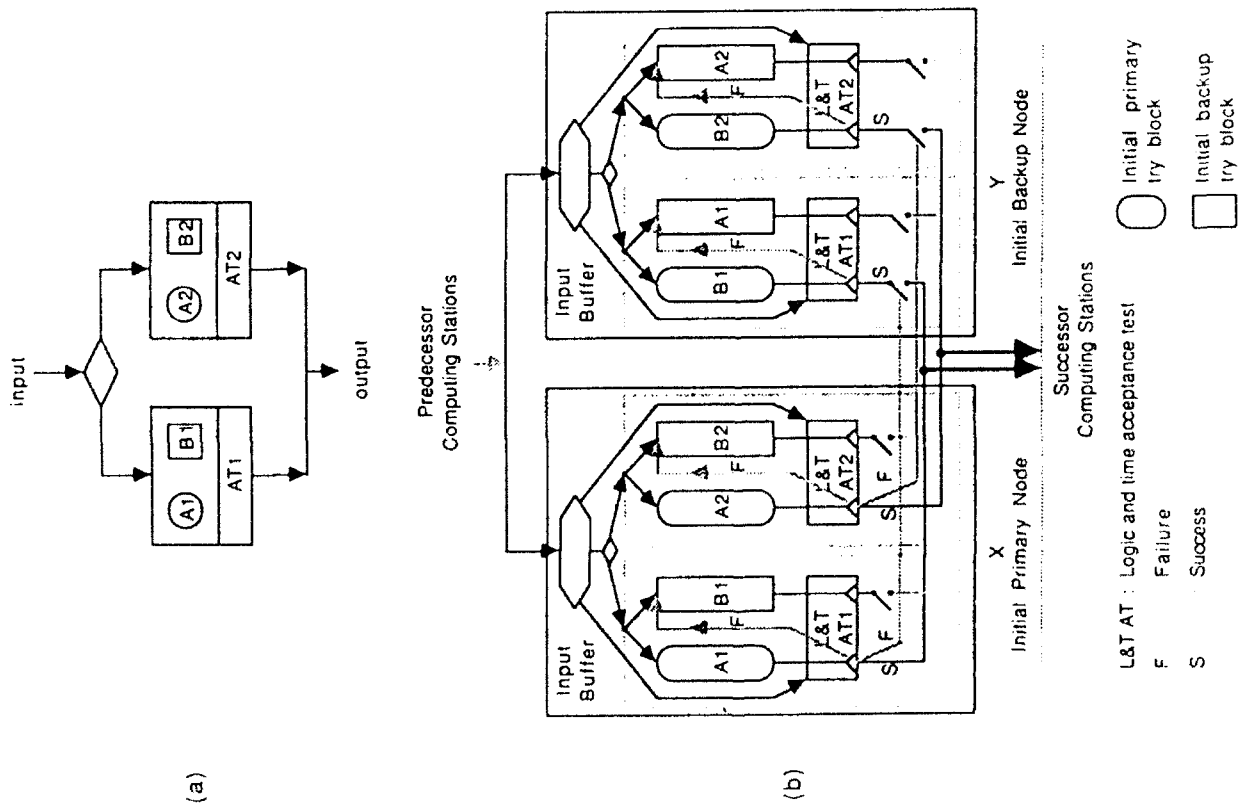


Figure 5. A multi-procedure or multi-phase DRB station

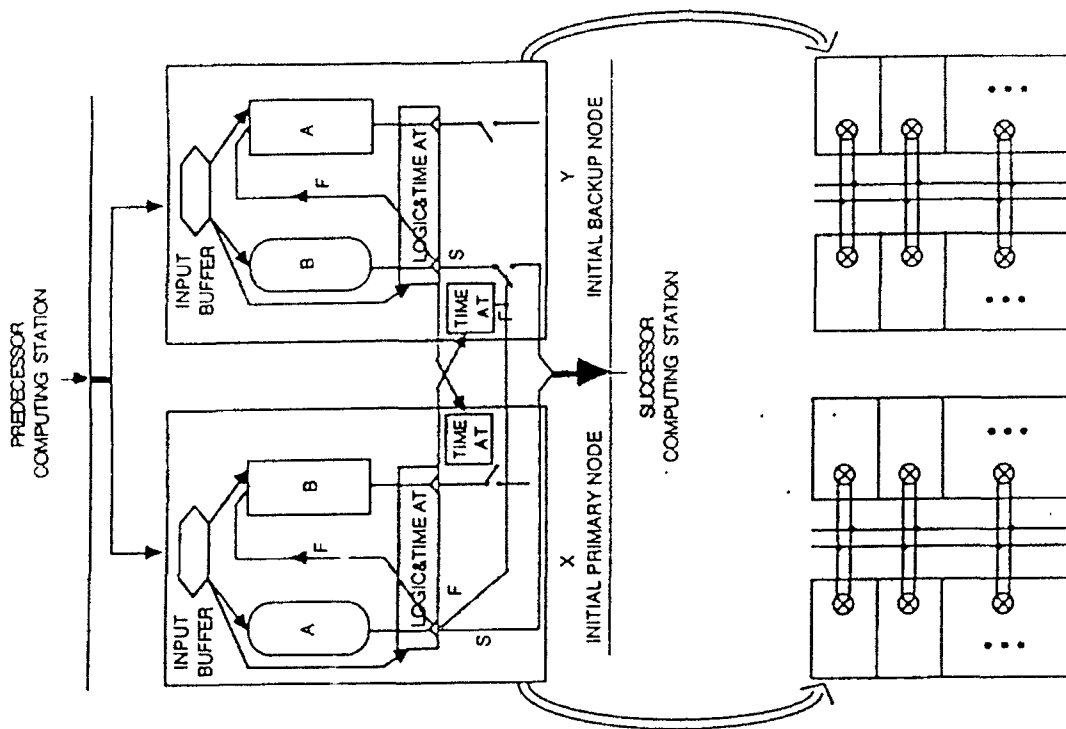


Figure 4. An extended DRB station incorporating the comparing processor-pair scheme

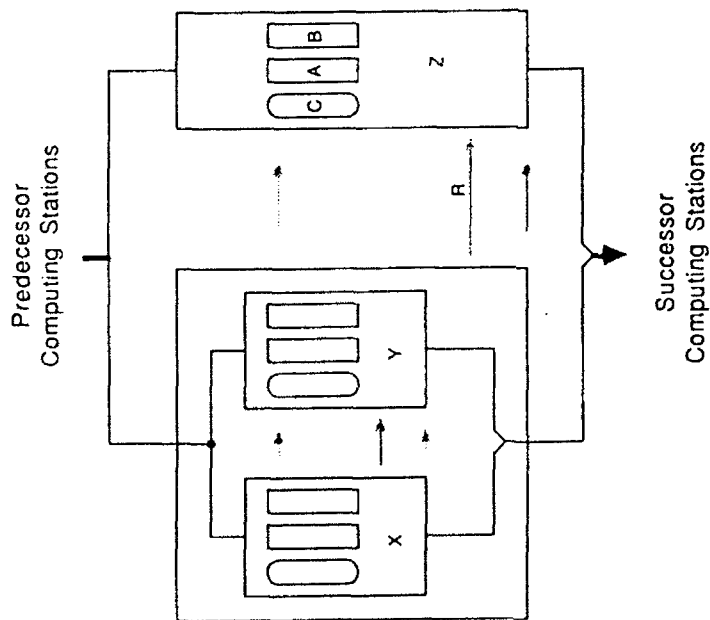


Figure 7. A DRB station that uses 3 try blocks
(Adapted from [Kim91])

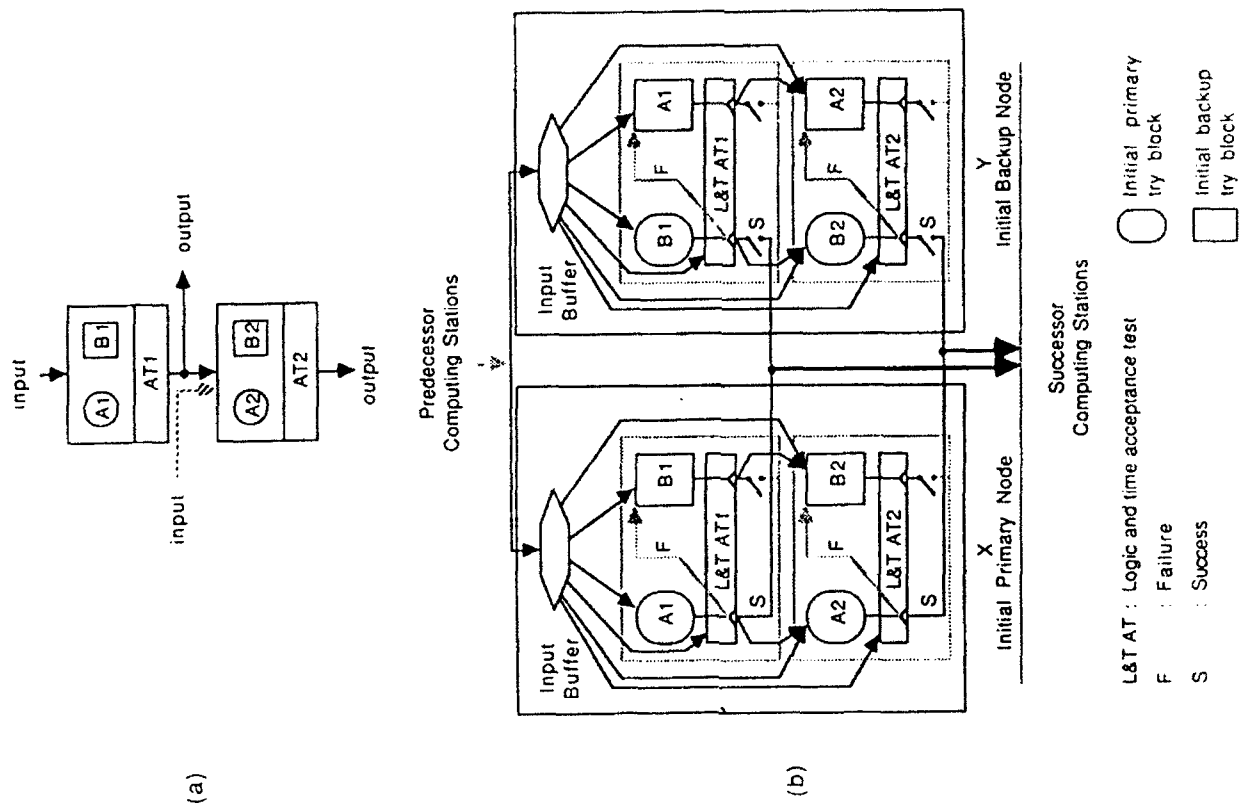
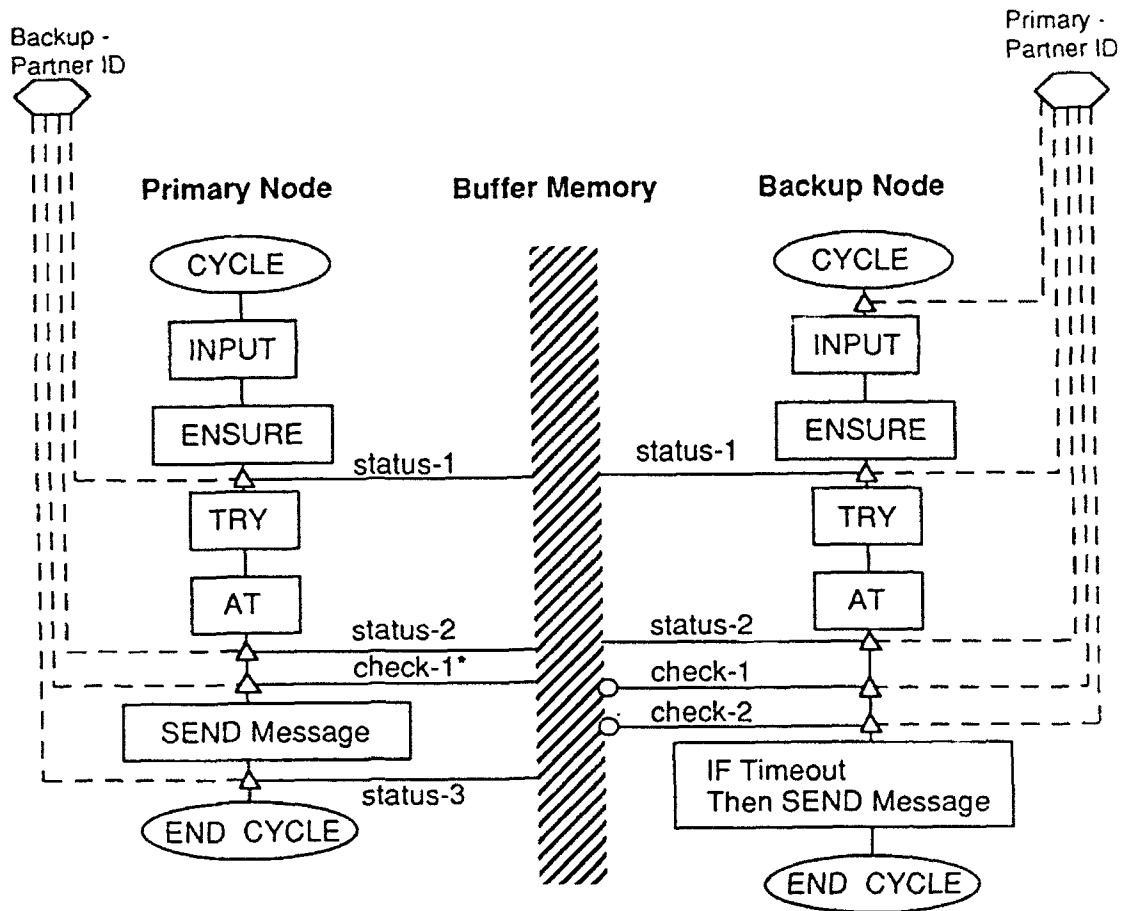


Figure 6. Serially bonded recovery blocks in a node within a DRB station



△ : The following actions are executed only in the DRB mode

○ — : Include possible wait

status-1 : Send node status message sequence number, main or backup node, primary or alternate try

status-2 : AT result

status-3 : Inform success of output delivery

check-1 : Check AT result of partner node with watch-dog timer on

check-1* : Check Progress/AT status of partner node

check-2 : Check delivery success of partner node with watch-dog timer on

Figure 8. An adaptive DRB station: Conditional activation of the DRB
(Adapted from [Kim92a])

REAL-TIME DEPENDABLE SYSTEMS DESIGN II

The BEAVER Program: A Tool for Survivability Analysis of Conceptual Distributed Systems

LT Clifford A. Whitcomb, USN
Ocean Engineering Department
Massachusetts Institute of Technology
77 Massachusetts Avenue
Cambridge, MA 02139

Dr. Deborah F. Allinger
The Charles Stark Draper Laboratory, Inc.
555 Technology Square MS 4E
Cambridge, MA 02139

**This paper is not included in the proceedings because it has not been approved for
public release.**

Application of the Hybrid Fault Model

Michelle McElvany Hugue*
Allied-Signal Aerospace Company
Columbia, MD 21045
michelle@batc.allied.com

Abstract

The single fault-type models employed in designing dependable systems usually provide either overly-optimistic or overly-pessimistic assessments of fault coverage or reliability. The mixed fault-type Hybrid Fault Model (HFM) permits more realistic algorithm design and associated system modeling. The HFM classifies faults in terms of the effects of these faults on system operations. The set of all faults is partitioned into three disjoint categories: *non-malicious faults*, *malicious symmetric faults*, and *malicious asymmetric faults*. Then, the type of algorithm required to detect or mask the subset of faults that is assumed to occur is indicated as a function of the fault type and a closed form expression for system reliability is provided. Reliability estimates for the hybrid model are then compared to those for existing models, and the impact of both models on system design decisions is assessed.

1 Introduction

The fault models used in designing dependable distributed systems typically make simplifying assumptions about the natures of faults¹ in the system. Often, the fault tolerance algorithms employed by a system treat all faults identically, ignoring the effects of any fault types the algorithm is not designed to distinguish or to tolerate. Such overly-optimistic single fault-type models assume a fixed number of benign permanent faults and perfect fault coverage. Or, the system model employs complex protocols that assume all faults to be pernicious, even though only a small portion of the faults may actually require such protection. By distinguishing different fault types and considering varying probabilities of occurrence of each fault type, we can develop more realistic system models to design algorithms capable of handling the various fault types.

We have previously defined the HFM and its impact on the reliability modeling of ultra-reliable systems [1, 2, 3]. In this paper, we examine the dependability and fault resiliency of several distributed system paradigms under the HFM, using the classical single-fault models as a basis for comparison. Under the HFM, the set of all faults is partitioned into three disjoint classes based on fault effects: *non-malicious*, *malicious symmetric*, and *malicious asymmetric*. Then, the type of algorithm required to detect or mask the subset of faults that is assumed to occur is indicated

*Supported in part by ONR Contract # N00014-91-C-0014

¹A *fault* is the identified or hypothesized cause of an error. An *error* is the manifestation of a fault, an undesired state either at the boundary or at an internal point in the system or process. A *failure* is the inability of the system or component to provide the specified service caused by an error.

as a function of the fault type. This matching of fault type to algorithm is important in ensuring adequate, yet cost effective, system fault coverage. If the fault tolerance techniques implemented do not support segregation and handling of mixed faults, then the hybrid fault model reverts to the overly optimistic or pessimistic single fault-type models, with no improvement.

After providing the motivation for our current work, we define the hybrid fault taxonomy on which the HFM is based. Next, the classical single-type fault models and their associated reliability expressions are presented in the context of our hybrid taxonomy. After an overview of the HFM and the associated reliability expressions, we define our dependable system framework. We next define several system characteristics, compare systems under both single-fault and hybrid fault models, and provide fault tolerance strategies appropriate to a variety of applications. After applying both fault models to example system paradigms, we conclude by summarizing our results and discussing future research plans.

2 Motivation

Several taxonomies have been proposed that provide the fault characteristics assumed by system fault and reliability models [4, 5, 6]. Characteristics such as duration (permanent, transient, intermittent); nature (hardware, software); behavior (arbitrary, restricted); and count (single, multiple) have long been used to model the assumed fault effects in computing or estimating system reliability [4, 7, 8]. With the exception of [9, 10, 11], the models also invariably focus on a single fault type. If the possibility of arbitrarily malicious or Byzantine faults [12] is considered, many fault tolerance algorithms and the resulting reliability estimates treat all faults as potentially Byzantine. Tolerant of such faults requires complex, communication-intensive protocols [12, 13, 14, 15], designed to restrict the malice of faults that can be introduced into the communication process. Thus, the arbitrarily malicious behavior of faulty nodes is prevented from disrupting the operation of non-faulty ones. Other fault tolerance strategies ignore fault types assumed to have low occurrence probabilities, such as Byzantine, and then adjust the reliability estimates using coverage factors [4, 7].

In defining the HFM, we assume a fully connected system consisting of *nodes* which communicate using synchronous message passing, with an upper bound on the time required for a node to generate and send a message. Individual nodes make decisions and compute values based on information received in messages from other nodes. The status of a node, faulty or good, is discerned by other nodes through the contents of messages originating from the target node, or through the lack of an expected message from that node. As in [9] and [16], a non-faulty node can always identify the sender of a message it receives and can detect the absence of an expected message.

3 The Hybrid Fault Taxonomy

The *hybrid fault taxonomy*, based on our work in [9], [11], and on the following definitions, classifies faults according to the errors they cause and the techniques needed to tolerate those errors.²

The *scope* of a fault refers to the portion of the system affected by that fault, also called the *fault extent*. A *symmetric fault* generates errors that are manifested identically throughout the

²Although we use a different definition of fault malice, these fault classes are equivalent to the classes of the same name in [9].

fault scope. An *asymmetric fault* generates errors that are manifested differently throughout the fault scope. Asymmetric faults are potentially more difficult to tolerate than symmetric faults.

Active redundancy techniques attempt to achieve fault-tolerance through fault-detection, alone or in conjunction with location and recovery. Since we are dealing with static redundancy management, no location or recovery techniques are addressed. *Passive redundancy techniques* use fault masking to hide the occurrences of faults and to eliminate the effects of faults, thus avoiding errors. For further details, see [4]. *Non-iterative* passive redundancy techniques require a single round of message exchange. *Iterative* fault masking techniques include procedures such as interactive convergence and interactive consistency, requiring additional rounds or iterations of message exchange among participants [13, 14, 16]. Fault-tolerant voting techniques, such as majority and median, are non-iterative passive redundancy primitives on which iterative passive redundancy techniques are often based.

Non-malicious faults can and will be detected³ in a non-faulty node by the active redundancy techniques implemented in that node. *Malicious* faults are those faults that cannot be detected by the implemented active redundancy techniques, but require masking using passive redundancy techniques.

Combining the attributes of fault malice and symmetry produces the four mutually exclusive and collectively exhaustive fault sets: *non-malicious symmetric faults* (B_S), *non-malicious asymmetric faults* (B_A), *malicious symmetric faults* (S), and *malicious asymmetric faults* (A). The *worst-case* (most severe⁴ or most difficult to detect or tolerate) faults are those in A , corresponding to the classic Byzantine fault where a faulty node supplies at least two different, correctly framed, values to different nodes. Faults in S are less severe than the faults in A , but are more severe than faults in $B_A \cup B_S$. Faults in B_S and B_A are comparable in severity, including benign faults, crash faults, and the subset of Byzantine faults that can be detected using active redundancy techniques, such as framing errors or missing messages. While the hybrid fault model, presented in §5, does not partition non-malicious faults into asymmetric and symmetric subsets, the single fault effects assumed in the classical models described below require this distinction.

4 Classical Fault Models and Reliability

Many systems fail to state their fault assumptions explicitly. Instead, they assume perfect fault coverage, even though the fault tolerance techniques they employ, which implicitly define an assumed fault model, may not be resilient to all fault types. A system using only active redundancy techniques is capable of detecting non-malicious or *benign faults* from set ($B = B_S \cup B_A$). However, if an (uncovered) malicious fault from set ($A \cup S$) occurs, system failure is likely to occur. When only non-iterative passive redundancy techniques, such as majority or fault-tolerant midpoint votes, are implemented, *symmetric faults* from the set ($S = B_S \cup S$) are masked, but the occurrence of *asymmetric faults* from set ($A = B_A \cup A$) can cause the system to fail. The use of interactive consistency and interactive convergence algorithms ensures all fault types are covered, since such algorithms mask arbitrary faults.

The assumed system dependability requirement is the ability to compute a correct result in the presence of faults, and we assume static redundancy management. Thus, combinatorial formulas

³By definition, a fault that is undetected by the active redundancy techniques implemented in a non-faulty node is malicious.

⁴Severity is subjective, relative to the context of the system model.

are sufficient to provide reliability estimates. Identical node reliability, $R(t)$, is assumed, although no assumptions are made about node failure distributions. The reliability is a function of m , the minimum number of good nodes required in a set of N nodes to maintain system operations, where R_m of $N(t)$ is given by

$$\sum_{i=0}^{N-m} \binom{N}{i} R(t)^N (1 - R(t))^i$$

and m is a function of the covered fault set [7].

Based on this discussion, we next define the classical fault scenarios C_B , C_S , and C_A , where the subscript is indicative of the faults covered by the model.⁵

C_B : Using active redundancy algorithms, a minimum of n_B nodes is needed to cover f_B faults in B , where $n_B = f_B + 1$. System reliability is given by expression (4) with $m_B = 1$.

C_S : Using non-iterative passive redundancy algorithms, a minimum of n_S nodes is needed to mask f_S faults in S , where $n_S = 2f_S + 1$. Reliability is given by expression (4) with $m_S = n_S = \lfloor \frac{(n_S-1)}{2} \rfloor$.

C_A : Using iterative passive redundancy algorithms, a minimum of n_A nodes is needed to mask f_A arbitrary faults, where $n_A = 3f_A + 1$. The reliability is given by expression (4) with $m_A = n_S = \lfloor \frac{(n_S-1)}{3} \rfloor$.

The impact of the classical scenarios on system design is addressed following the definition of the hybrid fault model.

5 The Hybrid Fault Model

The *hybrid fault model* comprises three scenarios based on the worst case faults covered in each scenario. Sets \mathcal{A} and \mathcal{S} are as defined in §3, while the set of non-malicious faults is given by $\mathcal{B} = \mathcal{B}_A \cup \mathcal{B}_S$. By definition, the sets \mathcal{B} , \mathcal{S} , and \mathcal{A} are disjoint. Since experimental evidence suggests that faults in \mathcal{B} are the most common, with faults in \mathcal{S} less common than those in \mathcal{B} , and faults in \mathcal{A} the least common of all, the fault assumptions made in a given system can be used to evaluate the impact of implementing the different HFM scenarios presented below.

5.1 Hybrid Fault Scenarios

The key to the hybrid fault model is to associate the proper hybrid algorithm with the assumed system or node fault set. The notation H_X is used to indicate that the scenario assumes that the worst case faults are in set X , where $X \in \{\mathcal{B}, \mathcal{S}, \mathcal{A}\}$.

H_B : Faults in $\mathcal{F}_B \equiv \mathcal{B}$ are covered. Hybrid active redundancy algorithms and at least n_B nodes are required to tolerate f_B non-malicious faults, where $n_B = f_B + (\tau_B + 1)$. The parameter τ_B is a fixed index, dependent upon the desired fault coverage, where $(1 + \tau_B)$ is the minimum number of nodes required for the system to remain operational.

⁵The subscripts B , S , and A , referring to benign, symmetric, and asymmetric faults, should not be confused with the sets \mathcal{B} , \mathcal{S} , and \mathcal{A} of the hybrid fault taxonomy. Although the set \mathcal{B} is equivalent to the set $\mathcal{B} = \mathcal{B}_A \cup \mathcal{B}_S$ as defined in §3, $\mathcal{A} \neq \mathcal{A}$, $\mathcal{S} \neq \mathcal{S}$, and sets \mathcal{B} , \mathcal{S} , and \mathcal{A} are not disjoint.

- H_S : Faults in $\mathcal{F}_S \equiv \mathcal{B} \cup \mathcal{S}$ are covered using hybrid non-iterative passive redundancy algorithms. At least $n_S = (f_B + f_S) + (\tau_S + 1)$ processes are needed to tolerate $(f_B + f_S)$ faults, where $S_{\max} = \lfloor \frac{n_S - 1}{2} \rfloor$ and $f_S \leq S_{\max}$. If operation in the presence of only one non-faulty node is possible, then $\tau_S = S_{\max}$. Otherwise, $\tau_S \geq S_{\max}$ if at least $(\tau_S + 1)$ good nodes are required.
- H_A : The fault set is $\mathcal{F}_A \equiv \mathcal{S} \cup \mathcal{B} \cup \mathcal{A}$; so, all possible faults are covered. Failure of the algorithm to tolerate any fault corresponds to a failure of the node running the algorithm. A minimum of $n_A = (2f_A + 2f_S + f_B + \tau_A + 1)$ nodes is sufficient to tolerate $(f_A + f_B + f_S)$ faults. The maximum number of faults in \mathcal{A} that can be tolerated is $A_{\max} = \lfloor \frac{n_A - 1}{3} \rfloor$ with $f_A \leq A_{\max}$, $\tau_A \geq A_{\max}$, and at least $(\tau_A + 1)$ good nodes assumed to be necessary for the system to remain operational. If a hybrid interactive consistency algorithm with r rounds of rebroadcast is used, then the further restriction of $f_A \leq r$ is also necessary.

The hybrid fault tolerance algorithms required by the HFM first apply an active redundancy technique to each message or value received by a node to discern any non-malicious faults, using, for example sanity checks, formatting checks, and error detection and/or correction codes. If a non-malicious fault is detected, such as a framing, parity, or encoding fault, a missing message, or a range violation, then a default error or status value is adopted as the value received by the node in the message. We can also assume perfect detection of non-malicious faults because any fault not detected by the active redundancy techniques implemented in the node is malicious by definition.

Next, passive redundancy techniques appropriate to the application are applied to the remaining values received by good nodes. We modify existing passive redundancy techniques to ignore or exclude the default error or status value from any calculations or comparisons. In the absence of non-malicious faults, no elements are excluded. Hybrid voting functions are derived in [1, 3] from the median, majority, and t-fault-tolerant mean and midpoint [14] functions used in non-iterative passive redundancy algorithms. A hybrid interactive consistency algorithm is presented in [9]. It should be noted that the exclusion of error values and the abilities of different nodes to receive different numbers of error values may result in a decrease in the number of values presented to the aforementioned fault-tolerant voting functions. Thus, the degree of fault tolerance of hybrid passive redundancy algorithms may need to be adjusted dynamically, as shown in [3] for a hybrid interactive convergence algorithm.

5.2 HFM Scenario Reliability

Since the assumed system dependability requirement is the ability to compute a correct result in the presence of faults, and static redundancy management is assumed, combinatorial formulas are sufficient to provide reliability estimates. The expressions for reliability under the HFM are more complex than those for the classical fault scenarios, as they are based upon the probabilities of occurrence of mixed fault types. The combinatorial formulas stated below are derived in [1, 11]. Again, identical node reliability, $R(t)$, is assumed, with no assumptions regarding the distribution of node failures.

Reliability under the three HFM scenarios can be estimated by considering a system's operational states under combinations of mixed faults. The conditional probabilities of occurrence of type \mathcal{A} , \mathcal{S} and \mathcal{B} faults, given that a fault has occurred, are given by μ_A , μ_S , and μ_B , where $\mu_A + \mu_S + \mu_B = 1$. Typically, we also have $\mu_B \gg \mu_S \gg \mu_A$.

Scenario H_A Reliability By definition, the number of nodes, N , satisfies

$$N \geq 2f_A + 2f_S + f_B + \tau_A + 1.$$

The system reliability is then computed by summing over all possible operational states according to H_A as follows:

$$\sum_{b=0}^B \sum_{s=0}^S \sum_{a=0}^A \binom{N}{b} \binom{N-b}{s} \binom{N-b-s}{a} \mu_B^b \mu_S^s \mu_A^a (1-R(t))^{b+s+a} R(t)^{N-b-s-a} \quad (1)$$

where $B = N - \tau_A - 1$, $S = \lfloor \frac{N-\tau_A-b-1}{2} \rfloor$, and $A = \min\{\tau_A, \lfloor \frac{N-2s-b-\tau_A-1}{2} \rfloor\}$.

For each operating state, the triple (a, s, b) , corresponds to the triple (f_A, f_S, f_B) , indicating the number of each type of fault occurring in that state. By varying the value of τ_A , this model covers both interactive convergence and interactive consistency algorithms. Equation (1) is equivalent to the expression given in [11] with $\tau_A = r$ under the assumption of a hybrid interactive consistency algorithm.

Scenario H_S Reliability By the definition of H_S , $N \geq f_S + f_B + \tau_S + 1$. If perfect fault coverage is assumed, the conditional probability of an asymmetric fault is taken to be zero, and we have $\mu_S + \mu_B = 1$. However, by assuming that μ_S and μ_B do not sum to one, i.e., $1 - (\mu_S + \mu_B) = \gamma_S$ for $\gamma_S > 0$, the probability of failure due to an uncovered fault can be included in the reliability computation. However, since the probability of system failure in the presence of a single fault in A is unity, no operating state can sustain an asymmetric malicious fault. So, regardless of the value of μ_A , we again sum over all the operating states to yield R_{sys} , given by

$$\sum_{b=0}^B \sum_{s=0}^S \binom{N}{b} \binom{N-b}{s} \mu_B^b \mu_S^s (1-R(t))^{b+s} (R(t))^{N-b-s}$$

where $B = N - \tau_S - 1$ and $S = \min(\tau_S, N - \tau_S - b - 1)$.

Scenario H_B Reliability We have $N \geq f_B + \tau_B + 1$, as defined in §5. If perfect coverage is assumed, then $\mu_B = 1$, with $\mu_S \equiv \mu_A \equiv 0$; otherwise, $1 - \mu_B = \gamma_B$ for some $\gamma_B > 0$, and the probability of correct operation in the presence of either a malicious symmetric or malicious asymmetric fault is zero. Thus, reliability under this scenario is given by

$$R_{sys} = \sum_{b=0}^B \binom{N}{b} \mu_B (1-R(t))^b R(t)^{N-b}$$

where $B = N - \tau_B - 1$.

6 System Fault Tolerance Strategies

As stated previously, we assume static redundancy management, where faulty nodes remain in the system; neither fault isolation nor reconfiguration is considered. Node failures are assumed to be exponentially distributed with failure rate $\lambda = 10^{-4}$ over a one hour mission.

N		2	3	4	5	6	7	8
C_B		1	2	3	4	5	6	7
C_S			1	1	2	2	3	3
C_A				1	1	1	2	2
H_S	(f_B, f_S, f_A)		$(\leq 2, 0, 0)$ $(0, 1, 0)$	$(\leq 3, 0, 0)$ $(1, 1, 0)$ $(0, 1, 0)$	$(\leq 4, 0, 0)$ $(\leq 2, 1, 0)$ $(0, \leq 2, 0)$	$(\leq 5, 0, 0)$ $(\leq 3, 1, 0)$ $(1, 2, 0)$	$(\leq 6, 0, 0)$ $(\leq 4, 1, 0)$ $(2, 2, 0)$ $(1, 2, 0)$	$(\leq 7, 0, 0)$ $(\leq 5, 1, 0)$ $(\leq 3, 2, 0)$ $(1, 3, 0)$ $(0, \leq 3, 0)$
H_A	(f_B, f_S, f_A)			$(\leq 2, 0, 0)$ $(0, 0, 1)$ $(0, 1, 0)$	$(\leq 3, 0, 0)$ $(1, 1, 0)$ $(1, 0, 1)$ $(0, 1, 0)$ $(0, 0, 1)$	$(\leq 4, 0, 0)$ $(\leq 2, 1, 0)$ $(\leq 2, 0, 1)$ $(0, \leq 2, 0)$ $(0, 1, 1)$	$(\leq 4, 0, 0)$ $(\leq 2, 1, 0)$ $(\leq 2, 0, 1)$ $(0, \leq 2, 0)$ $(0, 1, 1)$	$(\leq 5, 0, 0)$ $(\leq 3, 1, 0)$ $(\leq 3, 0, 1)$ $(1, 2, 0)$ $(1, 1, 1)$ $(0, \leq 2, 0)$ $(0, 1, 1)$ $(0, 0, \leq 2)$

Table 1: Classical and HFM Covered Faults

The main dependability requirement is that all non-faulty nodes compute "correct" values in the presence of faults, with the definition of correctness specified for individual scenarios. For simplicity, each node computes a data value, sends it to all other nodes, and decides on a correct value. All good nodes are expected to arrive at the same value.

Specific definitions for computing a correct final value determine the scenario that applies to the system model. In scenarios C_B and H_B , a node assumes its own value is correct, and compares this value to the values received from all other nodes, thus detecting the presence of faulty nodes in the system. In scenarios C_S and H_S , a node applies a fault masking algorithm to the set containing its personal value and the values received from all other nodes. The node adopts this voted value as the correct value. In scenarios C_A and H_A , each node executes an interactive consistency algorithm to achieve agreement among the non-faulty nodes upon the values sent by every node. A majority algorithm is then applied to the consistent value set to arrive at a final correct value.

We next present several design strategies based on this simple framework to further illustrate the implications of the HFM. These strategies are derived from the techniques described above for computing a correct final value under each scenario. The fault detection and masking algorithms used in the scenario are further specified to enhance the system tolerance to mixed faults. Several different assumptions about the types of faults and their relative probabilities are made. The characteristics of systems defined under the various strategies are then compared and contrasted by applying them to several examples in §7.

6.1 Strategy 1: Classical Single Fault Type Models

Our baseline strategy employs the basic single fault type models defined in §4. Table 2 presents the characteristics associated with each of the scenarios, with the number of faults tolerated given in Table 1 and the reliability given in Table 6 for all three scenarios for several values of N . The

Scenario	C_B	C_S	C_A
Fault Set	B	$B_S \cup S$	$B_A \cup A$
FT Tech	Compare	Masking	Masking
Active	Y	None	None
Passive	None	Majority	IC
Fault Prob	$\mu_B = 1$	$\mu_S = 1$	$\mu_A = 1$

Table 2: Strategy 1—Classical Models

main difficulties in using this strategy are the assumption of perfect fault coverage and the use of a single type of fault tolerance algorithm, implementing either passive or active redundancy.

We begin by examining C_B , as shown in Table 6 for up to 8 nodes. The reliability estimates obtained under this assumption are overly optimistic. Under this model perfect reliability is achieved for five or more nodes, even though, realistically, it may be impossible guarantee perfect fault coverage. Furthermore, computed values are merely compared and a detected difference is flagged. This model neglects the occurrence of malicious faults, and cannot guarantee that all good nodes will be able to recognize a correct value.

If scenario C_S applies, symmetric faults can be masked, and each good node will compute a correct value as long as the fault assumptions (type and number) are not violated. However, the implementation of a masking algorithm without dedicating additional resources to fault detection removes the system's ability to detect faults. Thus, if an uncovered fault occurs, the good nodes could potentially compute incorrect values with no indication of any fault. Again, the reliability estimates for C_S , shown in Table 6, are overly optimistic.

Unlike previous scenario estimates, the reliability estimate under scenario C_A is overly pessimistic, as it assumes that all faults are arbitrarily malicious. Perfect coverage to the number of faults shown in Table 1 permits the system to tolerate all types of faults. The implementation of interactive consistency or convergence fault masking algorithms permits the correct answer to be computed on all good nodes. However, faulty nodes are not detected. The effects of this model on system reliability are shown in Table 6.

6.2 Strategy 2—HFM with Perfect Coverage

We begin our discussion of the HFM under the assumption of perfect coverage. The specific characteristics of each scenario under the HFM are given in Table 3. Unlike the previous strategy, mixed fault tolerance techniques are employed, permitting detection of non-malicious faults in all scenarios. Masking of malicious faults in scenarios H_S and H_A ensures correct computations under that fault assumption. Framing checks are used to detect garbled messages, and then hybrid passive redundancy techniques are used to mask the remaining faults. The nominal values of τ are assumed, with $\tau_B = 0$, $\tau_S = S_{\max}$, and $\tau_A = A_{\max}$. The fault combinations that can occur for scenarios H_A and H_S appear in Table 1. Unreliability estimates under these scenarios under Strategy 2 are given in Table 6. The unreliability and faults covered for H_B are identical to those given for C_B . Although the possibility of uncovered faults is neglected in scenario H_S , the reliability obtained still exceeds that of the classical fault models, as can be seen by comparing the unreliabilities for the two scenarios in Tables 6. If scenario H_A applies, then the reliability estimates are improved by at least a factor of 10, as can be seen by examining the entries corresponding to H_A in Table 6

Scenario	H_B	H_S	H_A
Fault Set	B	$B \cup S$	$B \cup S \cup A$
FT Tech	Hybrid Detection	Hybrid Majority	Hybrid IC
Active	Framing, Compare	Framing	Framing
Passive	None	Hyb Majority	Hyb IC
(μ_B, μ_S, μ_A)	(1, 0, 0)	(.98, .02, 0)	(.98, .01, .01)

Table 3: Strategy 2—HFM with Perfect Coverage

Scenario	H_B	H_S	H_A
Fault Set	B	$B \cup S$	$B \cup S \cup A$
FT Tech	Hybrid Detection	Hybrid Majority	Hybrid IC
Active	Framing, Compare	Framing	Framing
Passive	None	Hyb Majority	Hyb IC
(μ_B, μ_S, μ_A)	(.98, (.02))	(.98, .015, (.005))	(.98, .015, .005)

Table 4: Strategy 3—HFM with More Practical Fault Coverage

and C_A in Table 6. The fault combinations given in Table 1 for the two scenarios are also valid for all strategies.

6.3 Strategy 3: HFM with More Practical Fault Coverage

As with the classical models in Strategy 1, the HFM scenarios in the previous strategy all assume perfect fault coverage. The reliability estimates reflect this assumption, with scenarios H_S and H_B achieving near-perfect reliability. Suppose the conditional probability non-malicious, symmetric malicious, and asymmetric malicious faults are assumed to be $\mu_B = .98$, $\mu_S = .015$, $\mu_A = .005$), as in Table 4. For scenarios H_B , the notation $(.98, (.02))$, used in Table 4, means that the probability of a covered fault is $\mu_B = .98$, with a probability of .02 that an uncovered fault will occur. The entries under Strategy 3 in Table 6 contain the unreliability estimates for both H_B and H_S , recomputed to account for the potential of uncovered faults which cause the system to fail, as described in §5.2. While these estimates may actually be somewhat pessimistic, they provide a lower bound on system reliability to complement the upper bound computed in the previous strategy. Since H_A covers all fault types, the corresponding unreliability estimates are identical to those under Strategy 2.

6.4 Strategy 4: HFM with More Fault Detection

A novel feature of the HFM is the ability to transform malicious faults into non-malicious ones by including more fault detection mechanism in the system. By definition, malicious faults are those whose effects can not be detected by the fault tolerance mechanisms implemented in the system. So, the inclusion of additional detection methods should decrease the conditional probability that a fault is malicious by increasing the types of faults that can be detected. The dependence of fault type on system factors is also a feature of the commonly used duration taxonomy, in that the distinction among permanent, intermittent and transient faults must be made relative the to the time granularity of the specific application and mission time. However, it is easier to add additional fault protection than to change the application or mission time parameters. Since the malice of

Scenario	H_B	H_S	H_A
Fault Set	B	$B \cup S$	$B \cup S \cup A$
FT Tech	Hybrid Detection	Hybrid Majority	Hybrid IC
Active	Framing, ECC Sanity, Compare	Framing, ECC Sanity, Compare	Framing, ECC Sanity, Compare
Passive	None	Hyb Majority	Hyb IC
(μ_B, μ_S, μ_A)	$(.99, (.01))$	$(a)(.99, .01, 0)$ $(b)(.99, .008, (.002))$	$(.99, .008, .002)$

Table 5: Strategy 4—Hybrid Fault Model with More Active Redundancy

		2	3	4	5	6	7	8
Strategy 1	C_B	1.0E-8	1.0E-12	9.7E-17	0	0	0	0
	C_S		3.0E-8	6.0E-8	1.0E-11	2.0E-11	3.5E-15	7.0E-15
	C_A			6.0E-8	1.0E-7	1.5E-7	3.5E-11	5.6E-11
Strategy 2	H_S		1.2E-9	2.4E-11	1.2E-14	1.7E-16	1.4E-17	0
	H_A			2.4E-9	4.1E-11	1.5E-11	4.2E-14	4.6E-16
Strategy 3	H_B	4.0E-6	6.0E-6	8.0E-6	1.0E-5	1.2E-5	1.4E-5	1.6E-5
	H_S		1.5E-6	2.0E-6	2.5E-11	2.0E-6	3.0E-6	3.5E-6
	H_A			2.4E-9	4.1E-11	3.8E-12	4.2E-14	4.6E-16
Strategy 4	H_B	2.0E-6	3.0E-6	4.0E-6	5.0E-6	6.0E-6	7.0E-6	9.0E-6
	$H_S(a)$		6.0E-10	6.1E-12	3.0E-15	0	0	0
	$H_S(b)$		6.0E-7	8.0E-7	1.0E-6	1.2E-6	1.4E-6	1.6E-6
	H_A			1.2E-9	1.0E-11	6.1E-13	1.1E-14	2.8E-17

Table 6: Unreliability for All Strategies

a fault depends upon the specific fault tolerance techniques implemented, the distinction between malicious and non-malicious faults can not be separated from the system design.

Thus, in this strategy, the additional active redundancy techniques employed in the system should cause the conditional probability of malicious faults to decrease. Instead of relying solely on framing checks to detect faulty node behavior in the form of garbled messages, error correction and detection codes can be implemented to permit information redundancy in data transmission to reduce the probability of an undetectably fault message. Sanity checks are employed to identify data values, in properly framed messages, that are outside the range of acceptable values. Comparison of values received from a node with the value obtained by applying passive redundancy techniques to a set of received data values permits detection of faults that were malicious in the previous strategy. Thus, the assumed conditional fault probabilities μ_A , μ_S , and μ_B can be adjusted to reflect the additional fault coverage, as shown in Table 5. The resulting increase in reliability is demonstrated in Table 6 under Strategy 4, where the values for H_S are computed with the perfect fault coverage assumption (a), and without it(b).

We next apply these strategies to two system design problems.

7 HFM Application to System Design

As evident from the strategies presented in the previous section, the classical single type fault scenarios contain few parameters that can be varied: the number of nodes, the number of rebroadcast rounds in the interactive consistency algorithm, and the number of good nodes required for system operation. The hybrid fault scenarios expand the parameters, permitting improved precision in modeling the system specified by the design requirements. The mixed fault types of the HFM provide dynamic fault tolerance, with linearly increasing system reliability as a function of increased system size. An additional advantage of the HFM is the use of fault segregation to justify design decisions. For example, to enhance the Byzantine fault coverage from 1 to 2 faults, the number of system nodes must be increased from 4 to 7. Under the Byzantine model, a system 5 or 6 nodes actually yields lower reliability than even a 4 node system (See Table 6, Scenario C_A). However, as we showed in [1] and [11], both the reliability and the system resilience to faults other than Byzantine increase when the HFM is used, justifying the use of additional resources without increasing algorithm complexity (See Table 6, Strategy 2, Scenario H_A).

To demonstrate the use of the HFM in making design decisions, we will examine several sets of design requirements, demonstrating why certain strategies and scenarios are most likely to achieve the design goals. Tradeoffs among the system cost, reliability, fault resilience, and system requirements are considered in defining candidate designs.

7.1 Example 1

Our first application requires a system of at most five nodes. All good nodes are required to compute a correct value and to flag the presence of some faulty nodes. The target unreliability is on the order of 10^{-7} . The probability of arbitrary faults is assumed to be negligible, and two faults must be tolerated.

We first examine the scenarios under Strategy 1 (§6.1). We can immediately reject C_B because good nodes are not guaranteed to compute a correct value in the presence of symmetric faults. While correct values will be computed in scenarios C_A and C_S and a four node system satisfies the reliability requirements, neither scenario detects the presence of even one faulty node. We therefore reject Strategy 1, and examine the strategies under the HFM.

Using Strategy 2 (§6.2), both scenarios H_S and H_A satisfy the correctness and unreliability requirements. Since hybrid algorithms are used, node faults that cause a message to fail the framing check will be detected. Since the probability of asymmetric faults is negligible, adopting the H_A scenario, with its interactive consistency algorithm, may not be cost effective. So, H_S with four nodes would appear to be adequate for this example, with its unreliability of $2.4E-11$ from Table 6.

However, the probability of an asymmetric fault, while negligible, is still non-zero. Thus, the reliability computed for H_S under Strategy 2, assuming perfect fault coverage, may be overly optimistic. The unreliability for this scenario under Strategy 3 (§6.3), with the probability of uncovered asymmetric malicious faults assumed to be .005, is given in Table 6 as $2.0E-6$. Thus, the reliability requirement is no longer satisfied. Furthermore, the system cannot detect potential corruption of the final value by an asymmetric malicious fault.

Although many system designers would still adopt H_S under Strategy 2, the HFM provides another alternative by permitting a further decrease in the probability of undetectable corruption due to an asymmetric malicious fault. In Strategy 4 (§6.4), additional active redundancy techniques

are implemented in the system. Messages are encoded and decoded with an ECC, sanity checks are applied to received data values, and the voted result is compared to all original values to detect the presence of an otherwise undetected fault. In this way, some of the asymmetric faults that were malicious under previous strategies are transformed into non-malicious faults under Strategy 4.

For example, the framing check under Strategy 3 would not have detected a value out of range in an otherwise correctly framed message. Unless all good nodes received the same incorrect value from that faulty sender, the sender has committed a (malicious asymmetric) fault which could cause the system to fail under Strategy 3. By implementing a sanity check to ensure that all values to be voted are within the correct range, Strategy 4 can mask faults of this nature. Once the final value is computed, the comparison of received values with the final value then permits a potential fault to be flagged. As shown in Table 6, the reliability of this system under scenario H_S in Strategy 4, without assuming perfect fault coverage, is estimated to be $8.0 \text{ E-}7$, which is in the desired range. If perfect fault coverage is assumed, the estimated reliability is $6.1 \text{ E-}12$.

Based on our analysis, there are several strategies which can be used in solving our example problem, permitting other factors to be addressed in choosing the final implementation. For example, scenario H_A could also be applied if the extra round of rebroadcast could be justified for some other reason, such as the elimination of a single point of failure. Another point of consideration is the requirement that two faults be tolerated. While the scenario H_S strategies we judged to be acceptable were all capable of tolerating two faults, no four-node implementation could tolerate two symmetric malicious faults. The ambiguity lies not in our model, but in the problem specification. A more precise definition of the types of faults to be tolerated could change the acceptable scenario and strategies appreciably, as evident in the next example.

7.2 Example 2

We now increase our reliability requirements and more precisely specify the faults to be tolerated. All good nodes are required to compute a correct value and to flag the presence of some faulty nodes. The target unreliability is now on the order of 10^{-10} . Up to eight nodes are permitted, and a minimum of two faults, one of which is arbitrary, must be tolerated. Other system considerations require that the weight and cost be minimized.

We first examine the classical single fault-type models of Strategy 1. The requirement that the system tolerate one arbitrary fault immediately removes scenarios C_B and C_S from consideration. Since the system must tolerate two faults, scenario C_A requires an 8 node system, with the reliability given in Table 6 as $5.6\text{E-}11$. To permit scenario C_A to detect some faults, a comparison function can be implemented to permit a node to detect differences between the final value and the original value sent by each node. With this modification, scenario C_A satisfies the requirements of this example.

However, if another strategy and scenario can be found which satisfy the same requirements with fewer system nodes, then the need to minimize cost and weight would make that scenario a more practical candidate. An examination of the properties of scenario H_A under either Strategy 3 (§6.3) or Strategy 4 (§6.4), given in Tables 4, 5, and 6, shows that the six node configuration satisfies the problem requirements. The choice of strategy could then be made based on the cost of implementing the additional active redundancy techniques used in Strategy 4.

8 Conclusion

In this paper we have presented an overview of both the classical single fault type models and the mixed fault type HFM. Using the hybrid fault taxonomy, we compared the fault tolerance of both classes of models. After providing closed form reliability expressions, we examined the reliability, fault resiliency and other system characteristics for a variety of system design strategies. We then applied the strategies to two examples, demonstrating the impact of the hybrid fault model upon the decision process in designing systems under a variety of constraints.

The major problem in applying this type of technique to system design or analysis is the need to estimate node failure rates and the conditional probabilities of mixed fault types. Thus, the comparisons provided in discussing the different system strategies should be applied according to the relative probabilities of various fault types, as few, if any, adequate techniques or data currently exist to provide precise estimates of the occurrences of mixed fault types. Another difficulty is ambiguous or incomplete system requirements, which do not provide an accurate representation of the desired system. This too represents an ongoing research problem that attempts to transcend the limitations of natural language in specifying systems.

We are currently extending our work to address dynamic redundancy management, requiring on-line fault detection, diagnosis, isolation, and system reconfiguration. Several facets of the diagnostic process which were not relevant under the static redundancy assumption need to be examined carefully. The detection mechanisms assumed in many of the strategies presented above will need to be expanded to lessen the ambiguity inherent in distributed decision making. Also, the importance of minimizing the diagnosis process bandwidth becomes more important than minimizing the amount of information exchange required to mask malicious faults.

The examples in this paper were kept simple to prevent the difficulties encountered in designing large, complex systems from hiding the decision processes supported by the HFM. However, many of the fault tolerance techniques implemented in the Multicomputer Architecture for Fault Tolerance (MAFT) system [17, 18] were chosen based on considerations similar to those we described. Our current work includes the application of the HFM to more realistic design problems.

References

- [1] M. M. Hugue, "The hybrid fault and reliability model for distributed systems." ONR contract technical report (submitted to rds92), ATC ASAC, Mar 1992.
- [2] N. Suri, M. M. Hugue, and C. Walter, "Reliability modeling of large fault-tolerant systems," in *Proceedings, 22nd Annual International Symposium on Fault Tolerant Computing*, p. (to appear), IEEE Computer Society, 1992.
- [3] M. M. Hugue and N. Suri, "Approximate agreement and the hybrid fault model," ONR contract technical report (submitted to rtss92), ATC ASAC, Mar 1992.
- [4] B. W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley Publishing Company, 1989.
- [5] J.-C. Laprie, "Dependability: Basic concepts and associated terminology." February 1990.

- [6] M. M. Hugue, "Fault type enumeration and classification," ONR Technical Report ONR TR 91-05, Allied-Signal Aerospace Company ATC, July 1991.
- [7] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Englewood Cliffs, N.J. 07632: Prentice Hall, first ed., 1982.
- [8] R. Geist and K. Trivedi, "Reliability estimation of fault-tolerant systems: Tools and techniques," *Computer*, vol. 23, pp. 52-61, July 1990.
- [9] P. Thambidurai and Y.-K. Park, "Interactive consistency with multiple failure modes," in *Proceedings, Seventh Symposium on Reliable Distributed Systems*, pp. 93-100, IEEE, October 1988.
- [10] F. Meyer and D. Pradhan, "Consensus with dual failure modes," in *Proceedings, Seventeenth International Symposium on Fault Tolerant Computing*, pp. 48-54, IEEE, July 1987.
- [11] P. Thambidurai, Y.-K. Park, and K. Trivedi, "On reliability modeling of fault-tolerant distributed systems," in *Proceedings, 9th International Conference on Distributed Computing Systems*, June 1989.
- [12] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *JACM*, vol. 27, pp. 228-234, April 1980.
- [13] D. Dolev *et al.*, "An efficient algorithm for Byzantine agreement without authentication," *Journal of Information and Control*, vol. 52, pp. 257-274, 1982.
- [14] D. Dolev, N. Lynch, S. Pinter, E. Stark, and W. Weihl, "Reaching approximate agreement in the presence of faults," in *Proceedings, Third Symposium on Reliability in Distributed Systems*, pp. 145-154, October 1983.
- [15] H. R. Strong and D. Dolev, "Byzantine agreement," in *Proceedings, 1983 IEEE Compcon*, pp. 77-81, IEEE Computer Society, 1983.
- [16] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, pp. 382-401, July 1982.
- [17] C. Walter, R. Kieckhafer, and A. Finn, "MAFT: A multicomputer architecture for fault-tolerance in real-time control systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, (Washington, DC), pp. 133-140, IEEE Computer Society, IEEE Computer Society Press, December 1985.
- [18] R. Kieckhafer, C. Walter, A. Finn, and P. Thambidurai, "The MAFT architecture for distributed fault tolerance," *IEEE Transactions on Computers*, vol. C-37, pp. 398-405, April 1988.

Design Capture for System Dependability

Jeffrey Zhou
Allied-Signal Aerospace Technology Center
9140 Old Annapolis Road/MD 108
Columbia, MD 21045
zhou@batc.allied.com

Abstract

It is essential to develop a set of system views which faithfully and completely specify complex computing systems from all important aspects, including such non-functional attributes as system dependability and performance. In this paper, we describe a dependability view which can be employed as a useful design tool to specify and analyze dependable systems. A real-time fault-tolerant operating system design is presented as a real-life case of using the dependability view.

1 Introduction

The development of a set of fundamental system views to capture all facets of a complex system design is essential in completely specifying such systems. Each system view should present an important aspect of the system and a complete set of views is needed to faithfully specify the system.

Five system views are commonly used in computer based system engineering (CBSE): Informational, Functional, Behavioral, Environmental, and Implementational [1]. The first three views are often used for system specification and design, while the last two views provide implementation constraints. The Informational View describes the system components and information flow among those components. Both system partitions and object relations are shown. The Functional View specifies the system functions and their inputs and outputs, describing the system operations and its response to stimuli. The Behavioral View describes different system states and their transitions, characterizing the dynamic system behavior. Each of these three views captures an important system aspect that is not covered

by the other two views. Since all three views portray the same system, relations among them can be exploited to ensure that the specifications are consistent. A mapping matrix can be used to establish the relation between the Informational View and the Functional View. Similarly, the state transitions in the Behavioral View can be traced in the Functional View by executing a series of functions. The current system development technology uses these three views, more or less independently, to capture system constructs. A number of computer-aided system engineering (CASE) tools have also been built based on these views. Nevertheless, these three system views, combined with the Environmental and Implementational views are not adequate to fully specify the properties of complex systems designed for distributed, real-time, and mission-critical applications.

In developing the Real-Time Executive Module (RTEM), an operating system kernel for real-time and fault-tolerant computing, we found that two essential views, the system dependability and system performance (real-time) views, should be added to the current set. In this paper, we concentrate on the dependability design of the RTEM system. RTEM is a complex, software operating system kernel which controls a distributed computer platform for real-time and mission-critical applications. In its original design document, all three available views were used to describe the system architecture, functions, and behaviors in high-level graphical diagrams. However, the system dependability had only textual descriptions and its specifications were embedded in the specifications of system functions used to implement redundancy management. Such textual descriptions are subject to different interpretations and usually present too much detail in the conceptual design stage. The mixed specifications also made the system dependability analysis less abstract and more implementation dependent. To overcome these shortcomings, a system dependability view needs to be defined independently in high-level system design to capture system dependability constructs.

In this paper, we discuss a method which is used in the RTEM development to construct a graphical representation for system dependability. We use two symbols, Fault Containment Region (FCR) and Redundancy Management Technique (RMT), as basic building blocks for constructing a system dependability view. A system can be partitioned hierarchically into multiple FCR's and each FCR contains some RMT's to define redundancy management techniques used in its region. We suggest that FCR partitioning and RMT selection be based on a hybrid fault model and its associated redundancy management techniques [2][3]. Although the dependability view proposed in this paper is not a formal one, it provides a framework for future

formalization.

2 Terminology

In this section, we introduce the terminology used in describing the dependability system view and the hybrid fault model. By *dependability*, we mean the qualitative property of a system that permits justifiable reliance on the services delivered[8, 6]. We use the standard definition of *reliability* as the conditional probability that a system is operating correctly throughout an interval of time, given that it was operating correctly at the beginning of that interval. The number of faults that the system can tolerate without becoming undependable is its *resiliency*.

Resource duplication or redundancy is used at the information, component, or computation levels to ensure fault-tolerance, correct operation in the presence of faults. *Active redundancy techniques* attempt to achieve fault-tolerance through fault-detection, alone or in conjunction with location and recovery. *Passive redundancy techniques* use fault masking to hide the occurrences of faults and to eliminate the effects of faults, thus avoiding errors. *Non-iterative* passive redundancy techniques require a single round of message exchange. *Iterative* fault masking techniques, such as interactive convergence and interactive consistency, require additional rounds or iterations of message exchange among participants [5]. Fault-tolerant voting techniques, such as majority and median, are non-iterative passive redundancy techniques on which iterative passive redundancy techniques are often based. *Hybrid redundancy techniques* combine active redundancy methods that detect faults with passive methods that mask the remaining faults.

Redundancy management is used to administrate the implemented redundancy techniques. If *static redundancy management* is used, faulty nodes or components remain in the system; neither fault isolation nor reconfiguration is considered. If *dynamic redundancy management* is employed, faulty nodes can be isolated and repair, recovery, or reconfiguration can be attempted.

Unlike previous work, we place no limitations on the duration of a fault; transient, intermittent and permanent faults are supported. The *hybrid fault model* classifies faults according to the errors they cause and the techniques needed to tolerate those errors, based on the following definitions. The *scope* of a fault refers to the portion of the system affected by that fault, also called the *fault extent*. A *symmetric fault* generates errors that are

manifested identically throughout the fault scope. An *asymmetric fault* generates errors that are manifested differently throughout the fault scope. Asymmetric faults are potentially more severe than symmetric faults.

Non-malicious faults can and will be detected in a non-faulty node by the active redundancy techniques implemented in that node. *Malicious* faults are those faults that cannot be detected by the implemented active redundancy techniques, but require masking using passive redundancy techniques.

Combining the attributes of malice and symmetry produces the three mutually exclusive and collectively exhaustive fault sets that make up the *hybrid model*: *non-malicious faults* (B), *malicious symmetric faults* (S), and *malicious asymmetric faults* (A). The *worst-case*, or most severe, faults in F are those in A . Faults in S are less severe than the faults in A , but are more severe than faults in B . The system dependability view presented below assumes the hybrid fault model is used in specifying the system's fault handling capacity.

3 System Dependability View

The system dependability view used in the RTEM development is a graphical notation which employs two symbols as well as their relations to describe system dependability. The two symbols are Fault Containment Region (FCR) and Redundancy Management Technique (RMT). An FCR is defined as a region beyond which a certain number and type fault cannot propagate. RMT is the technique used to fulfill the FCR objective. A system dependability design then can be described by using these symbols, either in a top-down or a bottom-up fashion.

If a top-down design method is adopted, a candidate system can be partitioned into several FCR's with coverage for the different fault types of the hybrid fault model. For example, a system may require that its data receiving subsystem to detect only non-malicious faults. Thus, this subsystem can be treated as one FCR. On the other hand, a central control subsystem executing the control logic for weapon launch may have to tolerate any type of fault including Byzantine faults [5]. This requirement defines another FCR with much stronger fault resiliency in its region. An FCR usually has a territory bound by natural hardware/software components. It also indicates its fault-tolerant capability with the number and types of faults tolerated in its region.

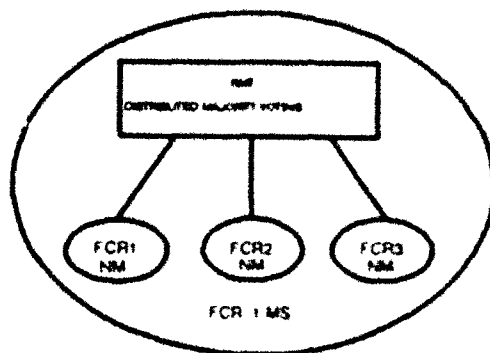


Figure 1: A simple dependability view

A system may have several top-level FCR's and each FCR can be unfolded to show a hierarchical structure. Figure 1 shows a simple dependability view with one layer of FCR hierarchy. The top-level FCR has three sub-level FCR's, each of which can tolerate only non-malicious faults. In other words, if a malicious fault exists, for instance in FCR1, it will not be detected/corrected and may manifest errors in the FCR1 outputs to the FCR region. If the top-level FCR is required to stop the fault propagation, or it cannot afford errors caused by the fault in its outputs, an RMT must be used to tolerate the fault. In figure 1, the RMT is a distributed majority voting algorithm, and the top-level FCR will be able to tolerate one malicious symmetric fault and multiple non-malicious faults.

It is interesting to note that the RMT implementation will affect FCR capability. For example, if the RMT in figure 1 is changed to the Triple Module Redundancy (TMR) technique, a single voter is introduced into the original FCR. It becomes a new fault containment region FCR4 as shown in figure 2. That changes the overall FCR resiliency. The FCR can no longer constrain the propagation of a malicious fault if it is originated in FCR4 and linked to FCR outputs. To preserve the original FCR dependability, FCR4 itself must be able to cover at least one malicious symmetric fault as shown in figure 3. The FCR4 then should have at least three voters and multiple links which perform cross voting. Please note that an FCR may change its dependability while still maintain its reliability. For instance, although the FCR in figure 2 changes its dependability, it may still be able to maintain its reliability if a highly reliable component is used for the single voter. While expressing system dependability explicitly, the Dependability

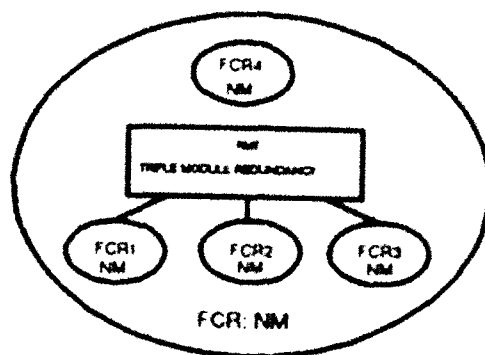


Figure 2: Dependability view with TMR

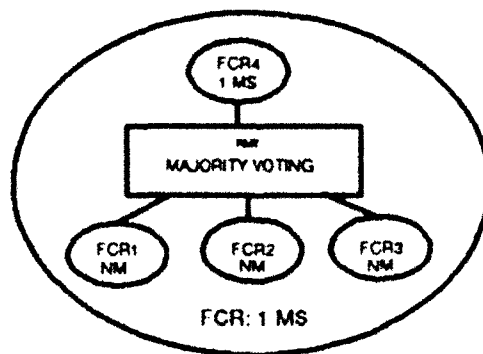


Figure 3: Multiple voters in a separate FCR

View also provides a mean for system reliability analysis [2].

4 Real-Time Executive Module

In this section, we present a real-life system design case to show how the Dependability View can be used to enhance design capability for fault-tolerant computer systems. RTEM is a software operating system kernel designed to control a distributed computing platform for real-time and mission-critical applications. The goal of our research is to develop a set of system executive functions which are portable to microprocessor based computing systems. These executive functions can be loaded to each processor in a distributed computing environment and perform both time management and redundancy management for a single node, as well as for the complex system. In the first phase of research, we are developing a concept-proof prototype which will be used as a flexible system model to test and verify design concepts for the Multicomputer Architecture for Fault-Tolerance (MAFT) [9]. In this paper, we concentrate on the discussion of system dependability design.

The dependability requirement for the RTEM prototype is to design a system that is able to tolerate a single fault of any type: non-malicious, malicious symmetric, or malicious asymmetric. In order to cover a single malicious asymmetric fault, the prototype needs a minimum four node configuration and a fully connected communication network to perform the Interactive Consistency Algorithm [4,5]. Figure 4 contains the Informational View of the system, which shows object partitioning and the information flow among components. Figure 5 illustrates the Functional View, in which seven main functions define the system operation. These functions

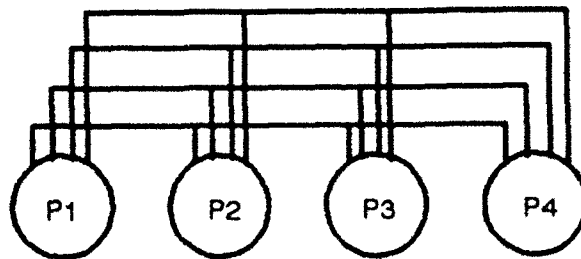


Figure 4: RTEM informational view

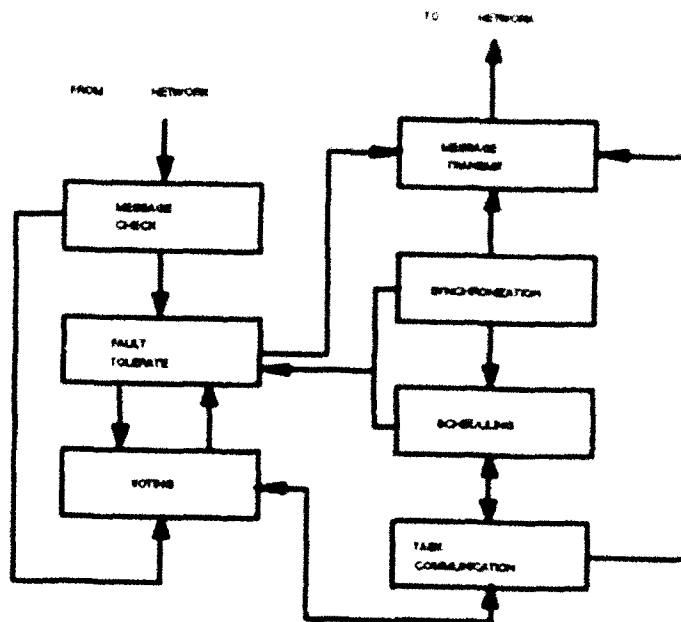


Figure 5: RTEM functional view

are loaded into each node in the distributed platform, and provide the core processes of time and redundancy management. If a faulty node is detected, dynamic redundancy management is required to reconfigure the system.

The system has three main operation states: cold-start, steady-state, and reconfiguration. The diagram of operation state transition is shown in figure 6. When system is powered up, it is in the cold-start state. During this period, all nodes try to synchronize with each other to form a common operating set. The Interactive Consistency Algorithm is computed so that all nodes can have a consistent common view about their membership in the

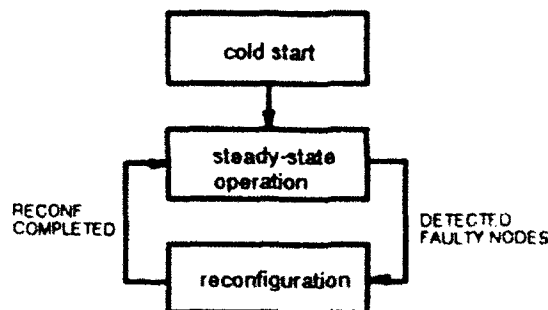


Figure 6: RTEM behavioral view

operating set. Once the operating set is formed, it triggers the transition from cold-start to the steady-state operation which is the main operation state for the RTEM system. The transition from steady-state to reconfiguration mode is triggered if a faulty node is detected by the system.

Constructing a dependability view is an attempt to establish a common language as a design aid for system engineers to capture, describe, and analyze fault-tolerant systems. The RTEM dependability view defines fault containment regions based on system partitioning and a fault containment architecture based on the system hierarchy. The dependability view also states the redundancy management techniques that are used, and how they are applied to tolerate faults in a particular region. The graphical representation provides a tool for system engineers to use common symbols and syntax to discuss system dependability design.

An FCR usually has a territory bound by natural hardware/software components. The entire RTEM system can be considered as one FCR which has the capability of tolerating any type of a single fault. In the FCR, there are several sub-level FCR's which provide necessary hardware/software redundancy to realize the fault-tolerant capability of the top-level FCR. The partitioning is based on hardware components and each processor can be naturally defined as a fault containment region. Since all four nodes are fully connected, the communication network can also be divided to four broadcasting channels and each of them can be included into its processor based fault containment region. Therefore, no separated sub-level FCR is used to represent the network component. In the following discussion, we will use FCR to refer the system level fault containment region. Component level fault containment regions will be referred as FCRI for an individual node or FCRs for multiple nodes.

After partitioning, we need to determine the fault-tolerant capability for each FCRI. There are a number of options, and the design choice must be made to meet dependability requirements for both components and the system. In RTEM, a node FCRI is not required to contain malicious faults. In other words, if a malicious fault is originated in an FCRI, it may appear in FCRI outputs. Nevertheless, it will not propagate through the system FCR region because of the redundancy techniques used among FCRs. Under this requirement, each FCRI has a single processor executing only active redundancy techniques for fault detection. Each FCRI includes as many as 25 different error detection mechanisms; so, a wide ranges of faults can be detected. A detected fault is then isolated and masked in the FCR region.

By the definition of fault containment region, FCRI errors can be moni-

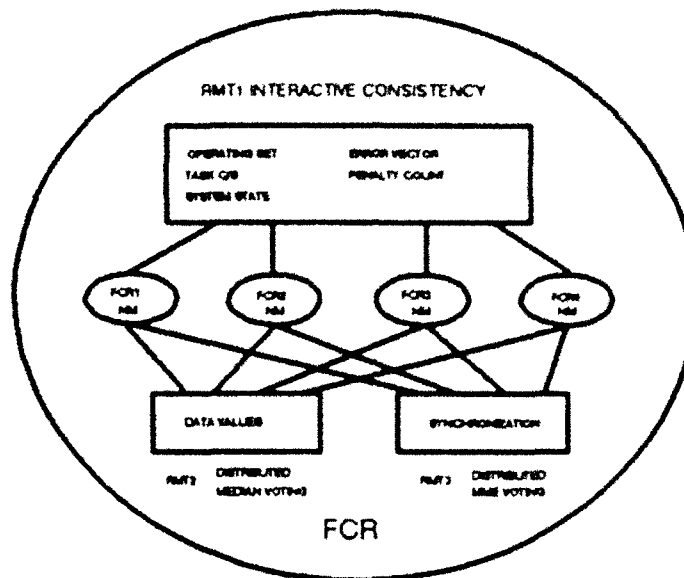


Figure 7: RTEM dependability view

tored only through message exchanges with other FCRs. A message can be a system state vector, task scheduling vector, application data values, etc. There are a number of messages flowing around the system. If a node becomes faulty, it may generate erroneous messages and broadcast them as the output of its FCRi. In order to fulfill the dependability objective of the system FCR, appropriate RMT, redundancy management techniques, should be applied to detect and correct those erroneous messages. The faulty node should be properly identified and then gracefully excluded from the system.

After defining FCRs and RMTs, we can construct the dependability view for the RTEM system as illustrated in figure 7. The view then can be used to discuss system dependability design. It is obvious that the key issue is to select adequate redundancy techniques for the system to tolerate a malicious asymmetric fault. If the system can cover a malicious asymmetric fault, it can cover a single fault of other types too. As shown in the dependability view, three different RMTs are used in RTEM. RMT1 computes the Interactive Consistency Algorithm for certain system messages so that all nodes will have a consistent common view for five important system data structures. They are: system operating set, task complete/start vector, system state vector, error vector, and penalty count for individual node. If a malicious asymmetric fault causes errors in these messages, it will be properly

masked by all good nodes in exactly the same way. Maintaining consistency is a necessary condition to achieve the RTEM design goal: a good node will never be commonly accused by majority nodes and a bad node will be commonly identified by all good nodes.

In the RTEM dependability view, system synchronization and application data are the only two messages which are not covered by RMT1 technique. Instead, they use majority voting algorithms to protect data integrity. Since a simple majority voting cannot detect or mask malicious asymmetric faults, the system may not be able to maintain the desired resiliency if such faults manifest errors in these two messages. Computing the Interactive Consistency check for all messages would be, of course, a safe design choice. However, the Interactive Consistency Algorithm is very expensive in terms of using network bandwidth. To cover a single asymmetric fault, the system needs two rounds of broadcasting and voting [5]. The first round broadcasts N copies of a message, and the second round rebroadcasts N^2 copies of the message. Then, the system is able to reach a consistent view about the message in N nodes by voting on the N^2 copies. The conservative design option is unacceptable to the RTEM system because large amount of application data will result in very poor system performance. A design trade-off has to be made between system dependability and performance.

Let us consider that an erroneous data value is generated by one node and broadcast differently to other nodes, which also receive good copies of the data from healthy nodes. If the erroneous data value has an acceptable deviance from good copies, it will not be excluded from voting. After voting, the system may keep different data values of the same data message in good nodes. That can be a malicious asymmetric fault. However, if homogeneous hardware and software is used for all nodes, computing results are kept the same for good nodes. Under this assumption, the simple majority voting is adequate to detect and mask malicious faults no matter it is symmetric or asymmetric. The shortcoming of homogeneous hardware and software is that the system is vulnerable to a generic fault. N -version designs and implementations are commonly used to cover generic faults. Nevertheless, it introduces deviance too. Since the goal of RTEM is to develop a set of system executive functions for a wide range of applications, we design the system in such a way that it supports both homogeneous and N -version applications. We leave the choices to application designers who should decide what type of fault-tolerant system they want to have. A similar idea is applicable to synchronization. A detailed discussion is beyond the scope of this paper and can be found in the reference [10].

5 Summary

The dependability view proposed in this paper is an attempt at constructing an independent system view in order to capture design for highly dependable systems. The two symbols, FCR and RMT, are powerful because they are closely associated with components and functions which are two basic building blocks in system design.

We are currently formalizing the definition and property of FCR for dependable system design. The partitioning of systems into physically independent segments has long been used to enhance system reliability. For an ultra-reliable redundant system, such partitioning is crucial to the system's dependability. Each independent segment is a natural FCR and is designed to limit the physical damage of a fault within a region to that region [11], and to localize the area in which fault recovery and repair are required [12]. The system dependability view is based on an extension of the definition of FCR to include the attributes necessary to define both physical and functional partitioning of hardware, software of functions, and to unify the concepts of fault and error containment.

We are also investigating a taxonomy of RMT's based on the hybrid model. The taxonomy will provide a systematic guidance for system engineers to select proper RMT's to meet system dependability requirement. The formal FCR, combined with the RMT taxonomy, will establish a foundation for constructing a formal system dependability view.

6 Acknowledgement

I would like to thank my colleague Dr. Michelle M. Hugue who provided terminology definitions and the hybrid model background for this paper. She also offered many constructive criticism and suggestions when the manuscript was prepared.

References

- [1] N.T. Hoang, "The essential views of system development," Proc. 1991 System Design Synthesis Technology Workshop, Silver Spring MD, Sept. 1991.
- [2] M.M. Hugue, "The hybrid fault and reliability model for distributed systems," Submitted to 11th Symposium of Reliable Distributed Sys-

terms.

- [3] M.M. Hugue, "Fault type enumeration and classification," ONR Technical Report, ONR-910915-MCM-TR9105, Nov. 1991.
- [4] P. Thambidurai and Y.K. Park, "Interactive consistency with multiple failure modes," Proc., IEEE Seventh Symposium on Reliable Distributed Systems, Oct. 1988, pp 93-100.
- [5] L. Lamport, R. Shostak, and M. Pease, "The Byzantine general problem," ACM Trans. on Programming Languages and Systems, Vol. 4, July 1982, pp 382-401.
- [6] B.W. Johnson, "Design and analysis of fault-tolerant digital systems," Addison-Wesley, 1989.
- [7] J.C. Laprie, "Dependability: basic concepts and associated terminology," Feb. 1990.
- [8] A. Avizienis and J.C. Laprie, "Dependable computing: From concepts to design diversity," Proceedings of IEEE, Vol. 74, May 1986, pp 629-638.
- [9] R.M. Kieckhafer, C.J. Walter, A.M. Finn, and P.M. Thambidurai, "The MAFT architecture for distributed fault tolerance," IEEE Trans. on Computers, Vol. 37, No. 4, April 1988, pp 398-405.
- [10] P. Thambidurai, A.M. Finn, R.M. Kieckhafer, and C.J. Walter, "Clock synchronization in MAFT," Proc. IEEE 19th International Symposium on Fault-Tolerant Computing, 1989, pp 142-149.
- [11] A.L. Hopkins, T.B. Smith, and J.H. Lala, "A highly reliable fault-tolerant multiprocessor for aircraft," IEEE Proceedings, Vol. 66, No. 10, Oct. 1978, pp 1221-1239.
- [12] D.P. Siewiorek and D. Johnson, "A design methodology for high reliability," The Theory and Practice of Reliable System Design, Edited by D.P. Siewiorek and R.S. Swarz, Digital Press, 1982, pp 621-638.

Using a POSIX Platform to Program Real-Time Concurrency and Time Fault Tolerance in Complex Systems

James Oblinger
Naval Undersea Warfare Center
Newport, RI 02841

oblinger@ada.nusc.navy.mil

Victor Wolfe
Department of Computer Science
The University of Rhode Island
Kingston, RI 02881
wolfe@cs.uri.edu

June 15, 1992

Abstract

Developing fault-tolerant real-time systems is complicated because they are often comprised of many special-purpose architecture components with non-standard interfaces. Furthermore, programming languages for these systems typically complicate the expression and enforcement of timing, concurrency, and fault-tolerance requirements. In this paper, we present the *RTC* programming environment that we are developing for the IEEE POSIX.4 standard interface. The *RTC* environment integrates explicit expression for timing and concurrency constraints and provides mechanisms for programming *time fault-tolerance*. We show how *RTC* run-time system enforces these constraints using the IEEE POSIX.4 interface. This use of a standard interface mitigates some of the complication present in designing fault-tolerant real-time applications.

1 Introduction

In *real-time* applications such as submarine command control and avionics, there are both timing constraints and shared resource consistency constraints that must be *predictably* met. These applications are often controlled by a distributed system to match the distributed topology of the components and to provide better performance through concurrency. Furthermore, many of these applications control delicate applications with severe consequences if these constraints are violated and left untreated. Programming such distributed real-time systems to meet timing constraints, consistency constraints, and provide fault-tolerance is a complex undertaking.

There are several current practices that add to the complication of developing fault-tolerant real-time systems. Most work in fault-tolerance for real-time systems has used special-purpose architecture¹ that incorporates techniques such as redundancy and voting [1, 2]. Using special-purpose architectures makes the problem of programming real-time systems difficult because it is usually not possible to take advantage of existing software tools. Also, software that is developed

¹In this paper we use the term *architecture* to refer to a system's underlying hardware and operating system

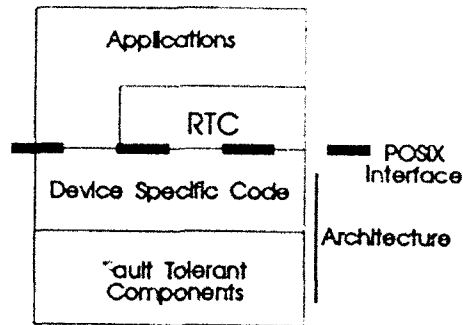


Figure 1: Design of Complex Fault-Tolerant Real-Time Systems

for such architectures is not usually portable or re-usable. This phenomena is depicted in Figure 1, where an extra layer of device-specific code is required to coordinate the various hardware components to realize an application. Since this layer is special-purpose and tailored to both the hardware and application, the system can not be easily modified or ported.

Another problem adding to the complexity of system involves the use languages such as Ada [3] for application programming. Ada requires that static priorities be assigned to tasks to “express” timing constraints. Since timing constraints are not explicitly stated, but are hidden in the relative priorities of tasks, constraints are difficult to write, verify and modify. Detecting and recovering from constraint violations is also complicated by the constraints being hidden. Other difficulties with Ada for real-time programming, such as its use of mutual exclusion with the possibility of priority inversion, are mentioned in [4, 5].

To address these difficulties in designing complex fault-tolerant real-time systems, we have designed the *RTC* software system that allows the explicit expression of timing, consistency, and reliability constraints in the application program, and supports their enforcement while executing on a *standard* interface to the architecture. The *RTC* programming language constructs support concurrent real-time programming by combining an abstract data type paradigm with a transaction-based paradigm while adding provisions for explicitly expressing timing and precedence constraints. The constructs are designed to be embedded in a host language; our current implementation is in C, but other host languages, such as Ada, can also be used. The *RTC* run-time system uses a special form of locking of shared resources, including processors, to allow a priority-based scheduler in the architecture to enforce the constraints expressed in the program without violating shared resource consistency constraints. To achieve portability and software re-usability among many widely-used architectures, we are implementing the *RTC* programming environment to execute “on top of” architectures that adhere to the proposed IEEE POSIX 1003.4a standard for real-time operating system interfaces. That is, the device-specific code is hidden under a standard POSIX interface, as shown in Figure 1. This design allows all application software to assume a standard interface to the

architecture, thus allowing software development that is independent of the underlying architecture and eliminating the need for device-specific code in the development of a complex application.

We use this *RTC*/POSIX interface to support fault-tolerance in the application software, instead of, or in addition to, fault-tolerance techniques in the underlying architecture. The form of fault-tolerance that we address in this paper deals primarily with *timing faults*, which occur when an application violates its timing constraints. Timing faults can occur for reasons such as a component failure or a transient overload. Since meeting timing constraints is required for correct execution in a real-time system, tolerance of timing faults involves ensuring that the system can achieve a *consistent state* (i.e. a state that meets safety requirements) when a timing fault occurs. For instance, if a submarine control program either misses its timing constraints or appears that it will miss its timing constraints, the control program should allow the system to recover without disaster.

To demonstrate these techniques, we will use a submarine application called MATE (Manual Adaptive TMA (Target Motion Analysis) Evaluator). The MATE application is an interactive display application that is used to compute a possible range, course and speed of a sonar contact. MATE consists of three major activities, the sonar input system, the MATE algorithm, and the display. The sonar input system simulates the output of a submarine sonar system and produces contact reports called Filtered Input Data Units (FIDUs) at a 20 second rate. The MATE algorithm uses FIDUs and operator selected range, course, and speed settings to produce a data point on the display. The processed data appears on the display as an unaligned vertical stack of dots. When the operator enters a new range, course or speed solution for the contact, the FIDUs are reprocessed by the MATE algorithm and displayed. When the dots are aligned vertically, the operator has found a possible solution. There are timing constraints on the input and generation of the output that must be met for correct performance. Concurrency control for shared resources, such as the FIDUs, must also be provided. We show how the *RTC*/POSIX system can be used to express these constraints, enforce them, and handle timing faults.

This paper is organized as follows. Section 2 presents the *RTC* language constructs and their use in the MATE application. Section 3 briefly describes the proposed IEEE POSIX real-time standard, and Section 4 describes how the *RTC* run-time system enforces the constraints expressed by the constructs by using a POSIX compliant real-time architecture. Section 5 summarizes strengths and weaknesses of our approach for supporting the development of complex real-time systems.

```

resource FIDU
    C declarations of data structures for FIDU stack

    action FIDU_read (FIDU_info)
        compatible FIDU_read
        C code for reading FIDU stack
    end action

    action FIDU_write (FIDU_info)
        C code for writing FIDU data structures
    except
        when ELABORT do restore FIDU to state before write end when
    end action

    /* resource body - initiation code */
end resource

```

Figure 2: Outline of *RTC* FIDU resource declaration

2 The *RTC* Programming Environment

The *RTC* programming environment consists of a set of language constructs that express real time concurrency constraints in a host programming language, and a run-time system that uses an underlying operating system to enforce these constraints. Our current implementation is embedded in the C language.

2.1 *RTC* Resources

RTC resource constructs provide abstract views of shared system entities such as devices and data structures. Each resource has private data structures and defines a set of *actions* that can be invoked by processes to examine or change the resource's private data. In the MATE example, there are several instances of resources including the FIDU Record and the Display. The FIDU Record requires concurrency control between the Sensor Processes which write to parts of the FIDU Record and the MATE process which reads from all of the FIDU Record. The Display resource requires that actions on it be *atomic* so that no incomplete or jumbled displays are seen. An outline of the *RTC* definition of a FIDU resource is shown in Figure 2. Each action specifies parameters for exchanging information with its invoking process and a **compatible** declaration to indicate permissible overlapping of execution of the action's execution that will preserve the resource's state consistency. For instance, in Figure 2, a *FIDU_read* action can overlap with with other *FIDU_read* actions. However, a *FIDU_write* can not overlap with any other action. Note that the compatibility

that can be specified is more general than typical read/write compatibility, although read/write is all that is illustrated in this example. The E_ABORT exception handler for actions shown in Figure 2 is called by the run-time system when the process that calls the action aborts its call (as discussed later).

2.2 RTC Processes

RTC processes constructs express how the application uses the resources. An example of the MATE process expressed with *RTC* constructs is shown in Figure 3.

Action Invocations. A process may invoke actions of resources *synchronously*, which causes the process to wait for action invocation to return, or *asynchronously*, which causes the process to continue executing. In the example of Figure 3, process *MATE* issues:

action Display.output(output_data)

to invoke the *output* action on the *Display* resource synchronously with parameter *output_data*. The call:

action& (FIDU_read_done) FIDU.read (FIDU_info)

invokes action *read* on resource *FIDU* with parameter *FIDU_info*. This call is asynchronous so that *MATE* does not wait for the action invocation to return before executing its next statement.

Events. In the asynchronous action call that we just discussed, *FIDU_read_done* is a predefined *RTC* type called *event*. An event has absolute time values that are established in three ways: 1) a **signal** statement by a process or action, which causes the current absolute time to be assigned to the event variable; 2) a **clear** statement by a process or action, which sets the event variable to an infinite absolute time; or 3) the run-time system signaling an event associated with the completion of an asynchronous action invocation (such as *FIDU_read_done*).

Timing blocks. *RTC* Timing blocks express earliest start times (**after**), latest start times (**before**), deadlines (**by**), and periods (**every**) for a series of statements using timing expressions involving event variables and relative times. A construct to express maximum execution time (**execute**) is also provided. Exception handlers for violations of many of the constraints expressed by these blocks can be expressed. There are several timing blocks in the example of Figure 3. One timing block states: **by last_update + 10sec**; the block expresses a deadline by which the part of the MATE process shown in Figure 3 must complete. Near the bottom of figure 3 is the E_DEADLINE exception handler that interrupts the constrained statements to execute if the deadline is violated. Another example of a deadline is: **by op_input**; where *op_input* is a global event signaled by the process monitoring the operator's track ball input. This deadline serves to interrupt the calculation

```

process MATE
:
event last_update, FIDU_read_done, OP_read_done, op_input

output_calculated = FALSE
by last_update + 10sec do
  guaranteed
  while (!output_calculated) {
    before (last_update + 8sec) do
      by op_input do
        exclusive
          action& (FIDU_read_done) FIDU.read (FIDU_info)
          action& (OP_read_done) operator.read (op_info)
        end exclusive
        after max (FIDU_read_done, OP_read_done) do
          calculate output_data
          output_calculated = TRUE
        except /* by op_input */
          when E.DEADLINE /* op_input */ do clear op_input end when
        end do /* by op_input */
      except /* before last_update + 8sec */
        when E.START do
          output_data = quick update
          output_calculated = TRUE
          warn operator of possible MATE malfunction
        end when /* E.START */
      end do /* before last_update + 8sec */
    } /* end while */
  no_except
    action Display.output(output_data)
    signal (last_output)
  end no_except
end guaranteed
except /* by last_update + 10sec */
  when E.DEADLINE alert operator of MATE malfunction end when
end do /* by last_update + 10sec */

```

Figure 3: Example of MATE Process with *RTC* Constructs

of the output data when new input data arrives; MATE then loops back and starts calculating the output data again.

Action Invocation Precedence Ordering. Flexible expression of precedence constraints (concurrency) within processes and between processes is supported using a combination of events, and synchronous and asynchronous action invocations, along with earliest start time constraints. For instance, concurrency within a process is expressed using asynchronous action invocations to “fork” off concurrent action invocations such as *FIDU.read* and *operator.read* in Figure 3. Then later in the process, the earliest start time constraint:

after max (FIUD_read_done, OP_read_done)

is used to “join” execution by requiring that, at this point, the process will wait for the two read actions to complete. By expressing fork and join semantics, *RTC* allows a powerful method for expressing concurrency within a process. Note that global events and *after* clauses in timing blocks can also be used to express precedence orderings between actions in different processes.

Exclusive blocks. The constructs **begin exclusive – end exclusive** denote a series of statements where all action invocations in the series must be executed on their resources without incompatible actions (as defined by the resource compatibility construct mentioned above) being executed while the block is active. In the example, an exclusive block is used to ensure that no incompatible action, such as a write to the FIDU, is allowed while the FIDU and operator input are being read.

Guaranteed blocks. The constructs **begin guaranteed – end guaranteed** denote a series of action invocations that must execute without delay due to contention for resources. In the example, the set of actions to read the inputs, calculate the output, and output the results appear within a guaranteed block. Immediately inside this guaranteed block there is a timing block with a latest start time constraint of *last_update + 8sec*. In this part of the process, we make the assumption that the actions take a maximum total of two seconds if there is no contention for resources. By constraining the actions with a latest start time of two seconds before the deadline and placing them in a guaranteed block so that there is no contention for resources, we know that the actions will only start if they will complete by the deadline (barring faults). This technique allows detecting potential deadline violations early so that they can be avoided. In the MATE process, if the actions are not started early enough, the E_START exception handler can generate a quick, complete, acceptable, but less accurate display.

No_except blocks. The constructs **begin no_except – end no_except** block exceptions for certain parts of the process. In the example, the call to *Display.output* and the updating of

the associated control information is done in a `no_except` block. This is done so that a deadline exception will not violate the atomic property of the *Display* device nor will such an exception leave the control variable of the MATE process, *last_update*, unsigned.

2.3 Timed Fault Tolerance: Support for Timed Atomic Behavior

The techniques that we use to handle timing faults are based on *atomic action* paradigms that have been widely used to support reliability in non-real time concurrent systems [6, 7]. In these traditional paradigms, changes to system resources and the environment are performed by actions (sometimes called *transactions*) with the property that, even if faults occur, either 1) the action completes and transforms the system to a consistent state, or 2) it appears that the action did not execute and the system is left in an original consistent state. That is, atomic actions perform all-or-nothing execution. Unfortunately, traditional atomic actions only require that all actions *eventually* decide whether to execute or not [7]; there is no deadline by which the decision and action must be completed. To incorporate the timing constraints of real-time systems into the consistency that atomic execution seeks to preserve, we would like to modify the definition of atomic behavior to allow all-or-nothing execution within timing constraints. Since meeting this criteria is provably impossible when faults [8] can occur, we alter our definition of timed atomic behavior further to be: *timed atomic behavior* is all-or-nothing execution within timing constraints or an exception. This definition is a generalization of the definition of timed atomic commitment presented in [9].

Specifying and Enforcing All-or-Nothing Behavior. To maintain consistency of the system it is often necessary to specify that either all actions in a set execute completely or none of them execute. To specify that all statements in a block complete, a `no_except` block can be used to delay exceptions until after the statements complete. Specifying the "nothing" alternative involves ensuring that no actions are executed if exceptions are possible during the `no_except` block. This is done by nesting the `no_except` block inside a `guaranteed` block as the first statement of a timing block. The timing block specifies a latest start time that is sufficiently far in advance of the deadline to allow the statements to complete when there is no contention for resources. The `guaranteed` block ensures that there is no contention for resources. If the statements are not started by this latest start time, the `E_START` exception is raised and none of the statements start; thus, achieving the "nothing" alternative. Note that the programmer must know the maximum execution time of the statements to be guaranteed in order to establish this latest start time. This technique is used to ensure the atomic update of the *display* resource in the MATE example (See Figure 3).

While this expression of "atomicity" is somewhat unconventional, the fact that real-time control applications directly affect the environment and are time-constrained makes traditional atomic rollback [7, 6] impossible. For example, if an action moves a robot arm from a starting position,

a compensating action [10, 11] can bring it back to the starting position, but not erase the fact that the move was performed or that the move took time. Thus, to achieve atomicity in a real-time environment, we require that either all of the constrained statements complete once they are started, or that none of them start.

Exception Handling. Since faults can occur, atomicity can not be guaranteed: *RTC* blocks therefore allow the expression of exception handling. We do not specify what the recovery actions are, but instead provide a general exception handling mechanism so that programmers can express various forms of recovery, including compensating actions [11, 10], imprecise computations [12], or other forms of roll-back or roll-forward techniques. As described in Section 2, blocks can specify exception handlers that interrupt execution for violation of latest start times, deadlines, maximum execution times and simultaneous execution. Furthermore, action declarations in *RTC* resources have an `E_ABORT` exception handler that becomes ready if the action invocation in the calling process is aborted. This exception handler allows the action to restore the resource to a consistent state; perhaps by employing a compensating action.

The `execute` timing block constraint be used to express a form of early detection of a potential deadline violation. If a fault causes a process or action to execute too long, it may eventually violate its deadline. However, by handling the execution time constraint before the deadline is violated, the deadline violation may be avoided. Consider the *RTC* code:

```
execute 5sec by NOW + 10sec do
  action r.a (params)
except
  when E_EXECUTE do early recovery actions end when
  when E_DEADLINE do recovery actions end when
end do
```

In this example, if a fault caused action *r.a* to execute for over 5 seconds, then the `E_EXECUTE` exception handler may be able to avoid a deadline violation that otherwise would have resulted from the fault.

For *hard* deadlines, which may be so critical that exception handling can not restore a consistent state if the deadline is missed, the *RTC* constructs can be used to specify an intermediate deadline that is sufficiently far in advance of the actual deadline to allow for recovery. For instance if a set of statements *S* has a hard deadline *D*, and restoring a consistent state from partial execution of *S* takes τ maximum execution time without contention for resources, the following timing block can improve the chances of being in a consistent state at *D* (let $D' = D - \tau$):

```

by D do
  by D' do
    statements S
  except /* D' violated */
    when E.DEADLINE do
      guarantee
        recovery (r time)
      end guarantee
    end when
  end do
except /* D violated */
  when E.DEADLINE do emergency actions end when
end do

```

Although this technique is pessimistic by raising exceptions when a violation may not occur in actuality, it improves the chances of being in a consistent state at D .

3 The POSIX.4 Standard Interface

Our approach to supporting the development of portable real-time software is to implement the *RTC* run-time system on an architecture that adheres to the proposed IEEE POSIX 1003.4a standards for real-time operating systems. We now present an overview of these proposed POSIX real-time standards.

Since 1985 the IEEE POSIX (Portable Operating System Interface) group has been developing an operating system interface standard [13] with the aim of standardizing the software interface over various architectures. They have also formed several subgroups, including one assigned to work on real-time extensions, now called 1003.4 [14], and one to work on a thread-based extension, called 1003.4a [15]². Although these extensions are not officially approved standards at the time of this writing, government agencies and industry have already demonstrated strong support for them. For instance, NASA has mandated that the software for the navigation system on its space station Freedom be POSIX.4 compliant.

Some of the features of particular importance in the POSIX.4 standard are:

- *Threads* – A POSIX.4 *process* is a unit of allocation for memory and devices. POSIX.4 decomposes a process into *threads*, each of which is a flow of control with its own program counter and run-time stack. In contrast to a Unix system where processes have a single thread, POSIX.4 allows programmers to take advantage of inherent parallelism in applications. Threads have minimal private state because they share state with all threads in the process. Threads can be scheduled individually, or a process can be scheduled and then locally determine which of its threads is to execute.

²Since this paper is concerned with real-time, we will use “POSIX.4” to mean that both of the POSIX 1003.4 and POSIX 1003.4a standards apply.

- *Real-Time Scheduling Interface* – POSIX.4 requires scheduling queues for a minimum of 32 priority levels, where threads from a higher priority queue are scheduled (preemptively) before threads from lower priority queues. Each thread may specify its priority and whether it is to be scheduled round robin or FIFO (round robin with infinite quantum).
- *Signals* – POSIX.4 requires at least 32 different signals. POSIX.4 signals are asynchronous messages between threads that contain very little data. Threads may send a signal, may wait on the arrival of a signal, and may have an asynchronous signal handler invoked upon arrival of a signal. POSIX.4 also supports real-time signals that can be queued at the receiver.
- *Timers* – POSIX.4 requires for system clocks and per-thread timers that can measure absolute and relative times on the order of nanoseconds. Timers can be “one-shot” or periodic, and can use either relative or absolute time; they notify threads of their expiration via signals.
- *Shared Memory and Mapped Files* – POSIX.4 allows the specification of main memory regions that can be mapped into the address space of multiple processes, allowing multiple processes to share memory. When one of the sharing processes writes to memory, all of the sharing processes can read it, thus allowing efficient sharing of data without explicit communication.
- *Message Passing* – POSIX.4 requires a message passing facility that includes provisions for queuing messages and several means of retrieving them. Message queues are system resources that are allocated to processes. A process can establish various attributes, such as queue length, for the queues that it owns. Asynchronous sends and receives of messages are supported.
- *Binary Semaphore and Mutex* – POSIX.4 requires binary semaphores as a means of synchronizing threads of different processes. It also requires *Mutexes* and *condition variables* as a means of synchronizing threads within in a single process. Mutexes, like semaphores, provide mutual exclusion among threads. However, mutexes can take advantage of the fact that they are local to a single process and can be implemented in the shared memory of that process. Mutexes and condition variables can be implemented with priority inheritance protocols [16]; this form of implementation is optional, but has been shown to be beneficial for preserving predictability in real-time systems [4].

These facilities are described in terms of a common interface that must be provided to applications. An exact implementation is not specified and, in fact, can vary as long as the required interface is provided. For instance, nothing precludes the architecture underlying the POSIX.4 interface from being a real-time architecture, such as SIFT [1] or MAFT [2], that includes traditional fault tolerance techniques such as redundancy and voting.

The POSIX.4 standard developers did not seek to describe every capability that should be provided by a real-time operating system, but instead provided a platform for building desired features. Therefore, it is not surprising that the capabilities described by the standard alone have deficiencies for supporting fault tolerant concurrent real-time systems. In particular, the standard does not require integrated support of real-time concurrency and reliability constraints such as: absolute timing constraints, exclusive execution constraints, and timed atomic behavior. For instance, POSIX.4 supports priority-driven preemptive scheduling that can be used to meet the timing constraints of processes, but the arbitrary preemption ignores the consistency constraints of resources. On the other hand, POSIX.4 supports mutual exclusion techniques to maintain the consistency of shared resources, but these techniques disallow potential concurrency and ignore timing constraints. Furthermore, the POSIX.4 standards also do not directly require fault tolerance support.

Although the POSIX.4 standard has these important deficiencies, we will show next that the capabilities mandated by the standard are sufficient for implementing the *RTC* run-time system which can support the development of real-time software that meets timing, concurrency, and reliability requirements. Since it appears that the POSIX.4 standard will soon be widely-used, building such a software system using the POSIX.4 standard should also enhance portability and re-usability of *RTC* based software that is developed.

4 Implementing *RTC* on POSIX.4 Compliant Systems

The constraints expressed by the *RTC* language constructs are enforced by a run-time system built on the commercial Lynx [16] operating system, which is POSIX.4 compliant. The *RTC* run-time system consists of a set of *managers* each of which is implemented using a POSIX.4 process and one or more POSIX.4 threads. Each *RTC* process is managed by its own *process manager* (QM), each *RTC* resource has a single *resource manager* (RM); and each processor has a *processor manager* (PM) which is used to reserve processors for guaranteed executions of actions. There is also a centralized *event manager* (EM) which interacts with process and resource managers to implement *RTC* events.

Scheduling. The Lynx operating system provides for scheduling queues at 256 priority levels which contain threads from all Lynx processes. Since reliable systems should be dynamic to allow for faults and other unforeseen occurrences in the environment, we wish to use a dynamic scheduling algorithm in the *RTC* run-time system. Preemptive Earliest-Deadline-First (EDF) scheduling algorithms have been shown to be optimal for meeting timing constraints in dynamic systems [17]. Simulating earliest deadline first scheduling with a priority-based system is possible [18], but requires infinite priorities (a priority for every possible deadline). We are currently investigating the

best assignment of threads to priority queues that best supports EDF scheduling. This assignment seeks to minimize the maximum length that any queue will reach because the maximum queue length is used to quantify the loss of performance, compared to the optimal EDF scheduling, that our *RTC/Lynx* system incurs. In addition to an original queue assignment, each thread must increase priority as its deadline nears. The exact details of this scheduling strategy are still being investigated.

Run-Time Support for Timing Blocks. To enforce the **after** construct of a timing block in *RTC* process q , the process manager task for q , QM_q , suspends q and uses the POSIX.4 timer and signal capabilities to request a signal at the earliest start time. When the signal arrives, process q re-activates. To enforce the **before** construct of a timing block in process q , QM_q requests that a POSIX.4 timer send a signal at the the latest start time. An exception handling thread waits for the signal. If the signal arrives before process q executes the statements of the timing block, the exception handler thread is activated and it deactivates the thread for process q . Otherwise, if process q starts the statements of the timing block, process manager QM_q deactivates the exception handling thread and removes the timer signal request. The **by deadline** construct of a timing block in process q is implemented using a stack to keep track of current deadline. As nested timing blocks are entered by process q , QM_q pushes the tighter deadlines on the stack; as the timing blocks complete, QM_q pops the deadline from the stack. Process manager QM_q uses the deadline on the top of the stack to determine the scheduling priority of the process, and to set a timer signal to indicate deadline violations. Again, a separate thread is used to wait for the deadline signal and then perform exception handling.

Resource and Processor Management. To ensure correct execution of an action, we need the ability to block incompatible action invocations. POSIX.4 provides *semaphores* and *mutexes* which block incompatible actions, as well as all concurrent actions, by using mutual exclusion. However, these mutual exclusion techniques disallow many concurrent accesses of a resource that would not violate consistency and as such they reduce utilization that could be valuable in meeting timing constraints. Instead of mutual exclusion, the *RTC* resource managers allow more potential concurrency through *semantic concurrency control* [19]. This concurrency control mechanism uses *action locks* at both the resource and processor level. At the resource level, if an action is locked, then compatible actions may execute, but no actions that are incompatible with the locked action may be executed until the lock is released. A process manager QM_q requests action locks from a resource manager RM_r by sending a POSIX.4 message specifying the set of actions that process q wishes to lock on resource r , $\{a_1, \dots, a_n\}$. RM_r grants the request only if each action in $\{a_1, \dots, a_n\}$ is compatible with resource r 's currently held action locks and pending requests of higher priority.

If RM_r does not grant a resource lock request, it queues the request based on process q 's priority. Resource managers use priority inheritance [4] by setting the priority of all currently executing threads for actions that are incompatible with the requested action lock to at least the priority of the requesting process.

Process manager QM_q can also request action invocations from an RM by sending the request in a POSIX.4 message. When RM_r gets an action invocation request a from QM_q that has not been locked, it must first lock the action. Once a lock is held for the action, RM_r creates a POSIX.4 thread t for action a and grants t access to the data of resource r through the POSIX.4 shared memory facilities. If process manager QM_q holds a processor lock, RM_r creates t with highest priority; otherwise, RM_r creates t with (requesting) process q 's priority. If process q 's action invocation is synchronous, process manager QM_q suspends q while waiting for return parameters from t ; if the action invocation is asynchronous, q is not suspended. When action invocation thread t completes, t sends the action's return parameters to a thread of the process manager QM_q in a POSIX.4 message. This thread shares memory with process q so that it can accept the returned parameters and update their values in process q 's state.

Meeting Constraints. To ensure exclusive execution, a process manager must obtain action locks for all action invocations in an exclusive block before it invokes any action invocations in the block. The action locks must be held until all action invocations in the block have completed. In this way, no action invocation that is incompatible with any action invocation in the exclusive block will overlap the execution of the block. To ensure guaranteed execution, a process manager must obtain action locks and a processor lock for all actions invoked in the guaranteed or simultaneous block before it invokes any action in the block. All locks are released when the block completes. The action locks ensure that no action invocation of the block is queued by an RM. The processor locks ensure that the action invocations execute on their assigned processors when the action invocations are ready and that the action invocations are not preempted.

5 Conclusion

This paper has described how the *RTC* programming environment can be used to naturally express real-time concurrency constraints in a C program so that the run-time system will enforce them on a POSIX.4 compliant architecture. This explicit constraint expression simplifies the development of the application software compared to the "expression" of constraints found in Ada. The *RTC* run-time system provides concurrency control through a locking mechanism that ensures the consistency of the shared resources. This mechanism potentially allows more concurrency with consistency than POSIX.4's and Ada's mutual exclusion capabilities. Once the locking mechanism determines which actions of a resource can execute, the run-time system employs the POSIX.4 preemptive priority-

based real-time scheduling for all threads (both actions and processes) in the system. Thus, the system integrates concurrency control and real-time scheduling. Moreover, the implementation of the *RTC* system using the IEEE POSIX.4 standard for architecture interfaces supports portability and re-usability of software across architectures that will ease application development. To handle timing faults, we also showed how the *RTC* constructs and run-time system can be used to enforce the requirement (at the software level) that either all of the constrained statements execute within timing constraints, or that none of them start. If faults occur, they are detected through exceptions and as exceptions.

A drawback of the *RTC* approach is the overhead incurred due to the managers in the *RTC* run-time system. Timing measurements of our preliminary implementation can be found in [5]. However, it is our belief that *predictable* performance is often more important than speed in a real-time system that must be reliable [20, 21].

We have used the *RTC* constructs to program distributed robotics applications that have real-time concurrency constraints, such as two arms that must coordinate to lift a moving object within timing constraints [5]. We are currently implementing the *RTC/POSIX* system and developing submarine applications, including MATE.

Acknowledgments. We thank Susan Davidson and Insup Lee of the University of Pennsylvania who were instrumental in the development of the *RTC* constructs.

References

- [1] J. H. Wensley, L. Lamport, J. Goldberg, M. Green, K. Levitt, P. Melliar-Smith, R. Shostak, and C. Weinstock, "Sift: the design and analysis of a fault-tolerant computer for aircraft control," *Proceedings of the IEEE*, vol. 66, pp. 1240-1255, May 1978.
- [2] C. J. Walter, R. M. Kieckhafer, and A. M. Finn, "MAFT: a multicomputer architecture for fault-tolerance in real-time control systems," in *Real-Time Systems Symposium*, IEEE Computer Society, 1985.
- [3] U.S. Department of Defense, "Ada Programming Language," 1983. ANSI/MIL-STD-1815A-1983.
- [4] R. Rajkumar, *Task Synchronization in Real-Time systems*. PhD thesis, Carnegie Mellon University, 1989.
- [5] V. Wolfe, *Supporting Real-Time Concurrency*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1991. Available as Technical Report MS-CIS-91-55.
- [6] B. Liskov, "Distributed programming in Argus," *Communications of the ACM*, vol. 31, pp. 300-312, March 1988.
- [7] P. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. New York: Addison Wesley, 1986.
- [8] J. Gray, *Operating Systems*, ch. Notes On Database Operating Systems, pp. 394 - 481. Springer-Verlag, 1979.

- [9] S. Davidson, I. Lee, and V. Wolfe, "Timed atomic commitment," *IEEE Transactions on Computers*, vol. 40, pp. 573-583, May 1991.
- [10] H. Tokuda, "Compensatable atomic objects in object-oriented operating systems," in *Pacific Computer Communication Symposium*, Oct. 1985.
- [11] A. Gheith and K. Schwan, "Chaosart: support for real-time atomic transactions," in *Nineteenth International Symposium on Fault-Tolerant Computing*, pp. 462-469, June 1989.
- [12] K. Lin and S. Natarajan, "Expressing and maintaining timing constraints in FLEX," in *Real-Time Systems Symposium*, pp. 96-105, IEEE Computer Society, 1988.
- [13] W. Corwin, C. D. Locke, and K. Gordon, "Overview of the IEEE POSIX 1003.4 real-time extension to POSIX," *IEEE Real-Time Systems Newsletter*, vol. 6, pp. 9-18, Winter 1990.
- [14] IEEE P1003.4 Working Group, "Real-time extensions for portable operating systems," 1989. draft.
- [15] IEEE P1003.4 Working Group, "Threads extension for portable operating systems," 1990. draft.
- [16] B. Gallmestier and C. Lanier, "Early experience with POSIX 1003.4 and POSIX 1003.4a," in *Real-Time Systems Symposium*, IEEE Computer Society, Dec. 1991.
- [17] C. L. Liu and J. Leyland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46-61, 1973.
- [18] I. Lee, R. B. King, and R. P. Paul, "A predictable real-time kernel for distributed multi-sensor systems," *IEEE Computer*, vol. 22, pp. 78-83, June 1989.
- [19] H. Garcia-Molina, "Using semantic knowledge for transaction processing in a distributed database system," *ACM Transactions on Database Systems*, vol. 8, pp. 186-213, June 1983.
- [20] J. Stankovic, "Misconceptions about real-time computing: a serious problem for next-generation systems," *IEEE Computer*, vol. 21, Oct. 1988.
- [21] J. Stankovic and K. Ramamritham, "What is predictability for real-time systems?," *Real-Time Systems*, vol. 2, September 1990.

METRICS

Developing a Metrics Assessment Program for the SLBM Software Development Division

William H. Farr, Annette Ashton

SLBM Software Development Division
Naval Surface Warfare Center Dahlgren Division
Dahlgren, Virginia 22448

ABSTRACT

This paper describes the ongoing effort in the Submarine Launched Ballistic Missile (SLBM) Software Development Division (K50) to establish a metrics program to cover the entire software life cycle process. This program will provide assessment information on both the division's software process and its products. In addition, this metrics program will provide a common terminology across the organization and a metrics database for all levels of the organization for present and future endeavors. This paper describes the objectives of this program, the approach used to identify a set of candidate metrics, and the selected set.

1. Introduction

To understand the role of a metrics program within the SLBM Software Development Division (K50) it is necessary to understand its mission and the types of software products it develops. The basic mission of this division is:

"To design, develop, produce, and maintain the shipboard **Fire Control Programs, Data, and Documentation** for all submarine launched ballistic missile systems, the associated support software (ex. compiler, linker, loader, etc.) and the software for effective targeting of the weapon system."

Notice from this mission statement that this division deals with three types of software: fire control, support, and targeting. Each poses both similar and unique problems in the various phases of the life cycle. All follow the fundamental waterfall chart for life cycle development and all follow a basic software framework that was established within the division. In addition, all three types have an established change control process as part of an overall configuration master plan. Each change control board, however, has slightly different software change control forms and problem reporting mechanisms. Each type of software follows a different standard for format and each type is developed by a different organization within the division.

In November of 1991, the division head, as part of an overall strengthening of our software development process, directed that a formalized metrics program be developed within the division that would cover all the different types of software and life cycle phases. He felt that "metrics" were being used in the software process and "kept" for historical purposes, but to varying degrees for the different types of software and life cycle phases. For the initiation of this metrics program, four basic objectives were established:

1. Identify all the metrics currently collected both informally and formally within the division for all the software types.
2. Determine whether the metrics currently collected are being effectively used for both assessment and control of our software process. If they aren't, develop a plan to accomplish that.
3. Determine what additional metrics need to be gathered and the implementation mechanism for areas in the process that need strengthening.
4. Develop a common terminology for metrics within the division, so that for a given metric there is an accepted definition/usage across the organization no matter what the type of software.

From these basic objectives and the structure of the division, some additional objectives were identified upon which the metrics program would be built. In the establishment of a program of this nature, a critical requirement is to form the objectives of the program **before** identifying the metrics to support it. One does not simply collect data without any objectives in mind! These additional objectives included:

- a. Concentrate on quality rather than productivity type metrics. We were concerned with both the quality of our development process and the resulting products. These metrics were to support the overall software process and management decision making at all levels.
- b. Use an iterative implementation approach. Implement and evaluate a few metrics at a time rather than establish a large data collection process requiring a lot of resources in time and personnel. We wanted to show the benefits of such a program and get support for it by all levels. The best way to do this was to demonstrate the benefit with a few initial metrics, gain acceptance with these, and then implement more in an iterative fashion.
- c. Involve all personnel in the development, implementation, and analysis of the metrics program. The primary reason such programs fail is a lack of commitment by the various levels of the organization. If all levels are involved in the shaping of the program, you'll take a big step in gaining support for it.

2. Approach

Our approach to establishing a metrics program was threefold:

1. Research work efforts within the division and to study software metrics, their applications and benefits to the overall software development process.

2. Establish a Metrics Study Group (MSG).

3. Implement a well-defined metrics selection process.

2.1 Research Work Efforts

The initial groundwork for the metrics program was laid by the authors. Particular areas to be researched were identified. These areas included work efforts accomplished within each branch and by the staff, how the waterfall software development process facilitates these work efforts, and how software metrics could be applied to these efforts.

2.2 Establish Metrics Study Group (MSG)

It was apparent that two people could not effectively accomplish all these tasks in a timely manner. The division head asked that each branch appoint a person to become a member or point of contact (POC) for the Metrics Study Group (MSG). Once the group members were appointed, the next step was to define MSG tasks, as well as POC responsibilities.

The MSG set the following as their tasks:

1. Define objectives for the metrics program

2. Research metrics concepts/applications

3. Identify areas of the software development process to which metrics should be applied

4. Define a specific set of metrics to be collected and implemented and develop a written metrics plan, to be updated as needed

POCs were chosen such that all three principal work areas were well-represented. There were individuals representing the development of fire control, support, and targeting software. Additional areas covered were data and documentation.

Also, each POC chosen had an understanding of how their organization operated within the software development process. They were not new employees just establishing themselves within the organization.

POC responsibilities were defined in two parts. Initial tasks were assigned to get the metrics program in place and then additional tasks were defined once the metrics were being collected/analyzed.

The POCs' initial tasks were to gain an understanding of software metrics and how they apply to their branch's work efforts. Several training sessions were held and reading material was given out and discussed at group meetings. Next, each POC set up an interview session with the authors and

respective selected key branch personnel. The purpose of this meeting was to uncover any metrics that the branch might already be collecting and using and to discover the areas in the development process that needed strengthening. Since the average branch member attending these meetings had not had any formal training in software metrics, a basic presentation on software metrics was given by the authors to tip-off the interview meeting. After the presentation, previously formulated discussion items were given to the meeting attendees to help stimulate discussion.

The MSG then analyzed the interview results from all branches and pointed out commonalities that occurred across branches within the division. Common areas were used as a starting point to define a candidate set of metrics which would be collected/analyzed as a part of the metrics program.

Once the candidate metrics are finalized, the POCs' task will be to set up data collection procedures within the respective branches. They will serve as focal points within the branches and the division on the data being collected. They will monitor data being collected and report to the MSG the effectiveness of this process. Analyzing data collected, reporting progress to management, training branch members (metrics awareness) about software metrics usage, and continuing to learn about software metrics are all important on-going responsibilities of the POC.

2.3 Metrics Selection Process

The metrics selected to be a part of the program were chosen based on two inputs. First, a prioritized list of areas in which to collect metrics was determined through the interview process. Second, a specific set of selection criteria was defined by the MSG. This was based upon impact, what other organizations were doing, and overall quality objectives of the division.

Branch interview sessions brought to light several key points taken into consideration in selecting the candidate metrics. It was clear that metrics were presently being gathered within the organization, but in a very ad hoc fashion. Little effort was being made to formally record past information which could be utilized to aid in making future decisions. Next, some branches used metrics more than others due to the nature of their work efforts. A third key point was that sizing and scheduling were common themes in most work areas. Finally, even though the division maintains a lot of different sets of computer software, few maintenance metrics were being gathered.

Six specific areas targeted for metrics were then defined. These were:

1. Scheduling, sizing, development manning
2. Better utilization of Configuration Management board information
3. Maintenance/reusability metrics
4. Design metrics such as complexity, size, speed, and modularity
5. Requirements development and stability
6. Testing metrics: unit/modular, integrated, system, and quality assurance

In conjunction with this prioritized list, the MSG also read and

discussed other organizations' metrics programs. As each area of interest was discussed, each of the following organizations was researched for which metrics they collected within that area:

- a. Army, Air Force, Navy
- b. Other major organizations at NSWCDD
- c. Professional Societies: IEEE, AIAA
- d. IBM*
- e. HP*
- f. MITRE
- g. NASA

*Please note that IBM and HP are two organizations noted by the Software Engineering Institute as having a level 5 rating for the Software Process Assessment Maturity Level. Having a rating of 5 indicates a very strong and mature metrics program.

Twelve quality factors are defined by [RADC,83] to improve the overall quality of software. Five of these quality factors were chosen as especially applicable to the work efforts within the division. The five quality factors chosen were Flexibility, Maintainability, Reusability, Testability, and Reliability. The metrics selected were chosen to help improve the software development process based on these quality factors. These particular quality factors were chosen due to the maturity of the division's software products.

Another important concern in developing a metrics program is the impact that it will have on the organization. A successful metrics program depends on the quality of data being collected. Asking personnel to change their way of doing their everyday job must be approached carefully.

The following criteria were also considered when the candidate metrics were selected:

- 1. Data availability
- 2. Implementation time
- 3. Required tools
- 4. Required training
- 5. Necessary changes to the division's software development process, including any changes to CM board policies and procedures

3. Proposed Metrics

This section lists the candidate metrics that have been proposed based on the selection criteria and the work that the MSG did as discussed in the last section. Table 1 gives a listing of those metrics, a brief definition and a group designation for the order in which the metrics will be implemented. Group 1 metrics were defined to be implemented within the first few months of establishing the metrics program. Groups 2 and 3 metrics will be implemented at later dates. As part of the final metrics plan that is being developed, each metric will have the following sections describing it: Definition, Benefit, Implementation, Quality Factor, and Type. The Definition will give the exact method for calculating the statistic. The Benefit section will give examples of how the metric can be employed and the resulting value. The Implementation section will discuss how the metric is to be collected, how

often, who will collect it, who will analyze it, how it is to feed back into the process, and what tools may be needed for implementation. The Quality Factor section will relate what Quality Factor the metric is associated with, while the Type section will relate what level the metric will be of benefit to (management, branch, or the software process).

It is intended that the list will be dynamic in nature. If a metric does not provide the benefit or information that is expected, other ones will be considered. If there are areas in the software development process that need additional metrics (it is anticipated that the requirements and design phases will fall in this category), new metrics will be added to this list. A good metrics program must be adaptable to a changing environment in order to be a viable part of the process.

Table 1. Candidate Metrics

METRIC NAME	METRIC DEFINITION	GROUP
KSLOC	Total number of executable lines divided by 1000.	1
KTLOC	Total number of all lines divided by 1000.	1
SLOCMOD	Total number of modules.	1
NUMPEO	Number of people assigned to a project per month.	1
PERMOD	% of modules that have changed.	1
#OF PR'S/PROGRAM	Number of problem reports (PR's) submitted against a program.	1
DEFECT DENSITY	Number of Software errors per KSLOC.	1
PR STATUS	Number of 'OPEN', 'ANSWERED', and 'RESOLVED' problem reports under configuration management.	1
REUSEMOD	% of modules that have been carried over from other software.	2
PMU	Program memory utilization.	2
MODCHG	% of modules that have changed from one baseline to the next.	2
DEFECT DENSITY/PHASE	Number of software errors per KSLOC per phase of life cycle.	2
ERROR URGENCY	% of software problems requiring an immediate fix over total number of software changes.	2
#OF PR'S RESULTING IN SCP	Ratio of number of PR's to Software Change Proposal's (SCP's)	2
ERROR CLASSIFICATION	Kind of software error that was made.	2
ERROR SEVERITY	Level of impact of the software error.	2
TESTCASE CLASSIFICATION	How the error was found.	2
DEPENDENCE	% of modules that call library, OS, or system routines.	2
CHANGE DENSITY	Ratio of number of SCP's over KSLOC.	2
NUMDOC	Number of documents associated with a given program.	2
DATABASE DEPENDENCE	% of modules with database references.	2
PDIM	% of errors introduced during the maintenance phase.	2
PDIE	% of errors resulting from enhancements to a program.	2
KSLOC/NUMPEO	Number of thousands of lines of code (developed, tested, etc.) per person per month	3
COUPLING	% of modules that call other modules.	3

TTF	Time to find and fix an error.	3
TTI	Time to identify the source of the error.	3
TTC	Time to determine a fix for the error.	3
TTV	Time to test the fix for the error.	3
PMOCS	% of modules violating coding standards.	3
SCP REASON	The reason why a software change proposal is necessary.	3
KPDL	Number of design lines for a program in the design phase.	3
# OF DOCUMENTATION PAGES	Number of pages within a specified document	3
REQUIREMENTS TRACEABILITY	Ratio of number of requirements at completion to number started with.	3
# OF REQ. CHANGES/PHASE	The number of requirement changes to the life cycle.	3

This set has been selected based upon the objectives and structure of the division and hence may not be applicable in its entirety for all programs. An organization must tailor its program based upon its resources, needs, and objectives.

4. Future Plans

The next step, after finalizing the metrics set, is to develop the implementation aspects of the metrics plan. Specific issues include who will collect the data, how often, who will analyze the data, what results are to be reported, and what data collection mechanism is to be employed. Also to be considered are the determination of the effectiveness of such a metrics implementation and what other metrics are to be considered for incorporation into the plan. Since the requirements and design phases were deemed important in both the conducted interviews and the overall objectives of the division, more metrics in these areas need to be identified.

This identification of specific areas and supporting metrics will be a continuing process. Once the division becomes accustomed to collecting and utilizing a specific set of metrics, then, as part of the iterative process, an additional set will be identified. The role of the MSG group in the future will be to help in this identification process and in the implementation of the supporting metrics.

5. Summary

In this paper we have described the effort undertaken within the SLBM Software Development Division to strengthen its overall development process for all of the different types of software that it produces. To do that we established certain objectives based upon management and division inputs. A committee of senior personnel from each branch within the organization was then set up to develop this metrics plan. Using inputs from the organization, the established objectives, and researching what other organizations had done in this area, a candidate set of metrics have been identified. The selected metrics were grouped into three categories based upon impact, data availability, and ease of implementation. Over the next year data will be collected, analyzed, and the results fed back into the process.

Key aspects in the development of this plan have been:

- 1) Establishing objectives, 2) getting participation from all levels of the

organization, 3) using an iterative approach for implementation, and 4) maximizing the use of data that is currently being collected. For this to be a viable program, getting the support of all personnel within the division is essential. To achieve this, ensuring their participation in developing the program and training them in the effective use of metrics is tantamount. By demonstrating the effective use of metrics in an iterative fashion, actively seeking their help and cooperation, and keeping the lines of communication open, we believe we'll have that support.

6. REFERENCES

[Conte, 86] Conte, S.D., Dunsmore, H.E., and Shen, V.Y., *Software Engineering Metrics and Models*, Benjamin/Cummings Publishing Co., Menlo Park, CA, 1986.

[Gilb, 77] Gilb, Tom, *Software Metrics*, Winthrop Publishers, Inc., Cambridge, MA, 1977.

[Grady, 87] Grady, R.B. and Caswell, D.L., *Software Metrics: Establishing A Company-wide Program*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1987.

[RADC, 83] *SOFTWARE QUALITY MEASUREMENT FOR DISTRIBUTED SYSTEMS*, RADC-TR-83-175, Volume I (of 3), Rome Air Development Center, July 1983

SOFTWARE QUALITY MEASUREMENT FOR DISTRIBUTED SYSTEMS Guidebook for Software Quality Measurement, RADC-TR-83-175, Volume II (of 3), Rome Air Development Center, July 1983

SOFTWARE QUALITY MEASUREMENT FOR DISTRIBUTED SYSTEMS Distributed Computing Systems: Impact on Software Quality, RADC-TR-83-175, Volume III (of 3), Rome Air Development Center, July 1983.

SYSTEM DESIGN FACTORS

Cuong M. Nguyen
Steven L. Howell

Advanced Systems Technology Branch
Underwater Systems Department
Naval Surface Warfare Center
Silver Spring, Maryland 20903-5000

ABSTRACT

The key to designing a real-time, large, complex system is to optimize the design to meet the requirements and desired measure of effectiveness. In order to achieve this, the system engineer/analyst must have the capability to specify the design goals/criteria, to quantify various aspects of the design, and to perform trade-offs among different design goals. One of the mechanisms that provides these capabilities is the System Design Factors. Whether the system design emphasis is on real-time, largeness, complexity, parallelism or any specific criteria, it requires a set of System Design Factor to describe the properties, attributes and characteristics of that system. Each System Design Factor must have its own metric to gauge every detail of that system. The metric describes the weaknesses and strengths of a specific area in the design. In turn, the correlation of the System Design Factor characterizes the completeness and robustness of the system. Whether the system is designed top-down, bottom-up, or middle-out, the System Design Factors have major influence in design capture and analysis, design structuring decisions, allocation decisions, and trade-off decisions between various design structures and resource allocation candidates.

The main objectives of the System Design Factors research are to provide a) A mechanism to communicate from the customer to the development team throughout various phases of system engineering, b) A mechanism to quantify and identify a large, complex, real-time system's strengths and weaknesses so that effective comparison of different systems is achievable and c) A mechanism for linkage of various aspects of the design, which help the system engineer or analyst to specify, capture, analyze, design, prototype, test, evaluate, trade-off and implement the system effectively. This paper presents a set of highly utilized System Design Factors that system engineers or analysts should consider early in the design to produce an effective system [HNH91], [HNH92].

KEYWORDS: System Design Factors, Structure Design Decision, Allocation Decision, Optimization Decision, Trade-off Decision, Large Complex Real-Time System.

1 INTRODUCTION

The way a system is traditionally built, starts with a customer who defines what is needed. These needs are analyzed to determine the requirements and specifications [EdH91]. In turn, these requirements and specifications are captured to produce the initial design [Hoa91]. Analysis is executed to assure that the initial design is complete and consistent [BIF90], [Hoa91]. This design is optimized iteratively until a feasible or optimal design is achieved [HNH91], [HNH92]. Collected results are then passed through for rapid prototype, assessment, evaluation, test and refinement to yield the final design [BoB85] [CYH91], [JeY91], [Kam91], [SvL76]. Implementation and test are then carried out to produce the final product, which is delivered to the customer. Many times, the customer will complain to the developer that the system did not meet the needs. The common causes for failing to meet the requirements might be one of the following: (a) the needs specified by the customer were not specific enough; (b) the needs were never clearly understood by the developers; or (c) communication among developers distorted the requirements as the development processes were performed.

The information understood by the whole system development team is crucial to produce the final product that meets the customer's needs. The current system engineering methodology lacks this communication mechanism from the customer to the whole development team.

The first objective of System Design Factors

research is to provide one such communication mechanism. In general, a system engineer or a customer wants some form to specify what criteria the end-result-system must meet. Depending on the desired criteria, it affects how the system would be designed and developed. These criteria are in turn the factors that the engineer must consider early in order to avoid bad designs, reduce cost, and optimize productivity [HHN90a], [HHN90b].

The second and third objectives are addressed by the following situation. Consider a situation where two system engineers were assigned to build a system independently given the same requirements and specifications from the same customer. When the two engineers delivered two systems to the customer, if the customer asks to compare quantitatively and qualitatively the different properties in term of performance, dependability, security, and real-time responsiveness of these two systems, then how does this comparison proceed. The second and third objectives of this research addressed this question. These objectives provide the mechanism for quantifying design goals of large, complex, real-time systems. With the current state of the system engineering technology, there are no normalized techniques to quantify and compare systems. If the system's properties could be specified quantitatively and qualitatively then its strengths and weaknesses can be identified and effective comparison among different systems can be achieved. Being able to qualitatively measure the system will not only benefit the system engineers for evaluation purposes, but it will also provide a benefit during the requirements specification phase, capture phase, analysis phase, design phase, optimization phase, and trade-off phase.

The proposed solution to these problems is to formulate hierarchical System Design Factors (SDF). The short term goal is to collect concepts and ideas from government, industry, and academic sources to formulate a complete and robust system specification. The individual factors will be studied independently. The correlation of factors will be investigated. Testings and applications will be made to verify the correctness and consistency of the formulation. The long term goals are to refine the formulation, provide automation, and provide new system engineering mechanisms and concepts that will have significant impact into the next generation of system engineering methodology.

The remainder of this paper is organized as

follows: Section 2: System Design Factors Taxonomy provides hierarchical view of SDF and provides current direction and focus of the research. Section 3: Example provides the touch and feel of SDF. Section 4: Specification and Use of SDF provides the utilization of the SDF template. Section 5: Current Status provides progress information; Section 6: Conclusion and Future Plans to provide on-going research pursuit.

2 SYSTEM DESIGN FACTORS TAXONOMY

The current thrust of this research is to define and formulate the System Design Factors and their relationship. These factors are categorized. The formulation of these factors expresses the relationship and behavior of closely and loosely associate factors. The effect of the individual factors on the design or engineering process is being studied. The correlation of multiple factors is also undergoing study. The rating, normalizing, and voting techniques for these factors are being derived. The research is expected to generate a robust SDF taxonomy. Each factor will consist of terminology, definition, source, metrics, example, usage, and notes.

Currently there are eleven major groupings of factors that seem to be required for most large, complex, real-time systems. These groupings are arbitrary. Each of the groupings consists of factors that are closely associated with other factors, which ultimately affect the factor's behavior by inheritance. This hierarchical taxonomy will evolve as this research effort progresses. The Current SDF taxonomy is shown below in Figure 1 (without any detail description due to the space permitted) to demonstrate the SDF framework. This taxonomy provides a set System Design Factors that customers, system engineers, or analysts should consider early in the design in order to produce an effective system [HNN91], [HNN92], [HHN90a], [HNN90b].

3 EXAMPLE

This section gives some small examples where SDF are used. Before this can begin, the characteristics structure must first be introduced. In order to effectively introduce the characteristics structure, some definitions are provided to give a common understanding.

1 PERFORMANCE		7.1.4 DEPTH	
1.1 RESPONSE TIME		7.1.5 AREA	
1.2 CAPABILITY		7.1.6 VOLUME	
1.3 RELATIVE ACTIVITY		7.2 WEIGHT REQUIREMENTS	
1.4 SPEED		7.3 RUGGABILITY (RUGGEDIZED)	
1.5 THROUGHPUT		7.4 SURVIVABILITY	
1.6 LATENCY		7.5 (PHYSICAL) PORTABILITY	
1.7 LOAD BALANCING		7.6 ENERGY REQUIREMENTS	
1.7.1 INFORMATION LOAD		7.6.1 (ENERGY) CONSUMPTION	
1.7.2 PROCESSING LOAD		7.6.1.1 ELECTRICAL (ENERGY CONSUMED)	
1.8 GRACEFUL DEGRADABILITY / LOAD SHEDDING		7.6.1.2 FUEL (ENERGY CONSUMED)	
1.9 EFFICIENCY		7.6.1.3 OTHER (ENERGY CONSUMED)	
1.10 PREDICTABILITY		7.6.2 (ENERGY) DISSIPATED	
2 REAL-TIME		7.7 LOCATIONAL OPERATING ENVIRONMENT	
2.1 HARDNESS		7.7.1 GEOGRAPHICAL LOCATION	
2.2 HARD DEADLINES	2.2.0.1 PERIODIC	7.7.2 INDOORS/OUTDOORS	
	2.2.0.2 APERIODIC	7.7.3 TEMPERATURE	
	2.2.0.3 SPORADIC	7.7.4 HUMIDITY	
2.3 SOFT DEADLINES	2.3.0.1 PERIODIC	7.7.5 ACOUSTICAL NOISE	
	2.3.0.2 APERIODIC	7.7.6 AIR PURITY/QUALITY	
	2.3.0.3 SPORADIC	7.7.7 EXPOSURE TO WIND	
2.4 TEMPORAL DISTANCE		7.7.8 EXPOSURE TO WATER	
2.5 TARDINESS		7.7.9 EXPOSURE TO ELECTROMAGNETIC RADIATION	
2.6 NUMBER OF CONSECUTIVELY MISSED DEADLINES		7.7.10 VIBRATIONS/STABILITY	
2.7 PREDICTABILITIES		7.8 CLIMATE CONTROL	
2.8 GRACEFUL DEGRADATION		7.8.1 COOLING	
3 COMPUTATION/PROCESSING REQUIREMENTS		7.8.2 HEATING	
3.1 IMPORTANCE		7.8.3 HUMIDITY CONTROL	
3.2 USEFULNESS		7.8.4 ACOUSTICAL NOISE SUPPRESSION	
3.3 PRIORITY		7.8.5 AIR PURITY/QUALITY CONTROL	
3.4 (COMPUTING) PORTABILITY		7.8.6 MOTION STABILIZATION	
3.5 INTERRUPT/RESET CAPABILITIES		7.8.7 LIGHTING	
4 DEPENDABILITY		7.9 MANUFACTURING CONSIDERATIONS	
4.1 RELIABILITY		7.9.1 PRODUCTION CAPACITY	
4.2 ACCURACY		7.9.2 PRODUCTION TIME	
4.3 FAULT TOLERANCE		7.10 COMPUTER	
4.4 GRACEFUL DEGRADABILITY		7.10.1 CPU	
4.5 REDUNDANCY	4.5.1 STATIC	7.10.2 MEMORY	
	4.5.2 DYNAMIC	7.10.3 STORAGE	
4.6 AVAILABILITY		8 FINANCIAL REQUIREMENTS	
4.6.1 INHERENT AVAILABILITY		8.1 COST TO DEVELOP	
4.6.2 ACHIEVED AVAILABILITY		8.2 COST TO PROTOTYPE	
4.6.3 OPERATIONAL AVAILABILITY		8.3 COST TO PRODUCE	
4.6.4 EASE OF REPLACEMENT		8.4 COST TO TEST	
4.6.5 CRASH RECOVERABILITY		8.5 COST TO PURCHASE	
4.6.6 COMPUTATION HEAVY PROCESS EFFECTS		8.6 COST TO OPERATE	
4.7 QUALITY		8.7 COST TO MAINTAIN	
5 SECURITY		8.8 COST TO REPAIR	
5.1 CLASSIFICATION		8.9 COST TO INCLUDE SECURITY CAPABILITY	
5.2 TYPE OF DATA		8.10 PRODUCTIVITY	
5.2.1 LEVEL I (CLASSIFIED)		9 TIME PROJECTED	
5.2.1.1 TOP SECRET OR ABOVE		9.1 ESTIMATED TIME TO DEVELOP	
5.2.1.2 SECRET		9.2 ESTIMATED TIME TO PROTOTYPE	
5.2.1.3 CONFIDENTIAL		9.3 ESTIMATED TIME TO PRODUCE	
5.2.2 LEVEL II (SENSITIVE)		9.4 ESTIMATED TIME TO TEST	
5.2.2.1 PRIVACY ACT/FINANCIAL		9.5 ESTIMATED TIME TO PURCHASE	
5.2.2.2 FOR OFFICIAL USE ONLY		9.6 ESTIMATED TIME TO OPERATE	
5.2.2.3 SENSITIVE MANAGEMENT		9.7 ESTIMATED TIME TO MAINTAIN	
5.2.2.4 PROPRIETARY/PRIVILEGED		9.8 ESTIMATED TIME TO REPAIR	
5.2.3 LEVEL III (NONSENSITIVE)		9.9 ESTIMATED TIME TO INCLUDE SECURITY CAPABILITY	
5.2.3.1 (OTHER-NOT CATEGORIES IN LEVEL I AND II)		10 LIFE CYCLE	
5.3 PERCENTAGE OF PROCESSING TIME / SECURITY LOAD		10.1 TESTABILITY	
5.4 ENCRYPTION TYPE REQUIREMENTS		10.2 MAINTENANCE	
5.5 IMPLEMENTATION TECHNIQUES REQUIREMENTS		10.2.1 EASE OF MAINTENANCE	
6 HUMANWARE		10.2.2 NUMBER OF PERSON NEED TO MAINTAIN	
6.1 EASE OF USE		10.2.3 NOTIFICATION	
6.2 POTENTIAL OPERATOR DECISIONS		10.2.4 FREQUENCY	
6.3 1. OPERATOR DELAY / 2. USER RESPONSE TIME		10.2.5 MAINTENANCE DOWNTIME/DURATION	
6.4 OPERATOR ACTION(S)		10.2.6 DEGREE OF SYSTEM DISABILITY	
6.5 1. REQUIRED NUMBER OF OPERATORS / 2. NUMBER OF SIMULTANEOUS USERS		10.2.7 WHEN MAINTENANCE COMES DUE	
6.6 USER INTENSITY		10.2.8 DURING MAINTENANCE	
6.7 AVERAGE TIME FOR EACH CATEGORIES		10.2.9 WEAR LIFETIME	
6.8 POTENTIAL ERRORS		10.3 OBSOLESCENCE LIFETIME	
7 PHYSICAL REQUIREMENTS		10.4 REUSE-ABILITY	
7.1 SIZE REQUIREMENTS		11 FUTURE NEEDS CONSIDERATIONS	
7.1.1 HEIGHT		11.1 ADAPTABILITY/FLEXIBILITY	
7.1.2 WIDTH		11.2 EXPANDABILITY	
7.1.3 LENGTH		11.3 COMPATIBILITY	
		11.4 INTERGRABILITY	
		11.5 INTEROPERABILITY	
		11.6 INTEGRITY	

FIGURE 1. TAXONOMY OF SYSTEM DESIGN FACTORS

Quantitative Value is a quantifiable measurement. It is a numerical value. It represents the degree of excellence. Some value may have different type of range or minimum and maximum

cardinality. For example, temperature could be measured as 120.5 degree of Kelvin and could vary only between 0 and 277.15 degree. **Attribute** is the quality of a person or thing (non-physical)

Property is the attribute which belongs to some one or some thing (physical).

Characteristics is any special feature of a person or thing.

The hierarchical relationship among these definitions forms a characteristics structure which provides a general mechanism for quantification. This mechanism is applied with the System Design Factors to quantify systems. An example is given to demonstrate the relationship among these definitions. The example in Figure 2 shown hierarchically a Subject has Properties which have Attributes which in turn have Quantitative Value and Qualitative Value (Characteristics). Consider an eagle who has the following properties: Performance, Life-Cycle, and Physical Requirements. Performance which in turn have the following attributes: Air Speed, Land Speed, and Take Off Time. Life Cycle which in turn has the following attributes: Overall Sickness Time (health) and Life Span. Physical Requirements which in turn have the following attributes: Size, Color, and Wing Span. Size which in turn has the following quantitative value (i.e., could vary between 0.5 to 2.0 feet) and qualitative value (characteristics) (i.e., could be small, medium, or large). The rest of the quantitative and qualitative values are shown in Figure 2.

Subject	Properties	Attributes	Quantitative Value	Qualitative Value (Characteristics)
Eagle	Perf.	Air Speed	10 to 50	Slow
			51 to 75	Moderate
			76 to 100	Fast
		Land Speed	1 to 5	Slow
			6 to 10	Moderate
			11 to 15	Fast
		Take-Off T.	0.0 to 10.0	Fast
			11.0 to 20.0	Moderate
			21.0 to 30.0	Slow
	Life Cycle	Sickness T.	0 to 1	Good
			1 to 3	Average
			6 to 10	Poor
		Life Span	0.0 to 5.0	Short
			5.1 to 10.0	Medium
			10.1 to 15.0	Long
	Phys. Req.	Size	0.5 to 0.75	Small
			0.76 to 1.5	Medium
			1.6 to 2.0	Large
		Color	1 to 5	Brown
			6 to 10	Gray
			11 to 15	Black
		Wing Span	1.0 to 10.0	Short
			11.0 to 15.0	Medium
			16.0 to 20.0	Long

FIGURE 2. EXAMPLE OF CHARACTERISTICS STRUCTURE

In the above example, the Subject was an eagle. However, the Subject can be substitute with one of the following: system, subsystem, component,

module, task, node, device, or any object. This characteristics structure provides a low level or detailed link to the criteria which in turn provides a high level link to the System Design Factors. In other word, the characteristics structure applied to eagle to allows us to quantify and rate different aspects of its species. This similar approach can be applied to the system there by allows to quantify and rate different factors of the system. The application of the characteristics structure to the System Design Factors is demonstrated in Figure 3.

Subject	Properties	Attributes	Quantitative Value	Qualitative Value (Characteristics)
or		or	or	
Subject	Factors (Goal Oriented)	Associated Factors (Criteria/Decision Oriented)	Quantitative Value	Qualitative Value (Characteristics)
System, Subsystem, Component, Object, etc.	Perf.	Resp. Time	0.0 to 1.0	Fast
			1.1 to 2.0	Medium
			2.1 to 3.0	Slow
		Throughput	1 to 5	Slow
			6 to 10	Medium
			11 to 15	Fast
		Latency	2.1 to 3.0	Slow
			1.1 to 2.0	Medium
			0.0 to 1.0	Fast
	Depen.	Reliability	0.5 to 0.8	Bad
			0.9 to 1.0	Good
		Fault-Tol.	0.0 to 1.0	Bad
			1.0 to 2.0	Moderate
			2.0 to 3.0	Good
		Weight	0.0 to 10.0	Light
			11.1 to 20.0	Medium
			20.1 to 30.0	Heavy
	Phys. Req.	Size	10 to 50	Small
			51 to 100	Medium
			101 to 150	Large
		Power	0.0 to 1.2	Low
			1.3 to 2.5	Moderate
			2.6 to 3.8	High

FIGURE 3. EXAMPLE OF SYSTEM DESIGN FACTORS

As illustrated in this example (Figure 3), a customer may need to rate, measure, or design the system in term of the following Properties: Performance, Dependability [Joh85], [WaH91], and Physical Requirements. Performance which in turn has the following Attributes: Response Time, Throughput, and Latency. Dependability which in turn has the following Attributes: Reliability and Fault-Tolerance. Physical Requirements which in turn has the following Attributes: Weight, Size, and Power. The Response Time could vary between 0.0 to 3.0 and its characteristics could be fast, medium, or slow. The rest of the quantitative and qualitative values are shown in the graph. This mechanism allows one to identify and effectively compare the strengths and weaknesses of different systems.

4 SPECIFICATION AND USE OF SDF

The example in Figure 3 shows the overall or top level application of the SDF. The detail application of SDF is demonstrated through The System Design Factors Template (Figure 4). The purpose of this template is to provide a general format to guide the system engineer or the customer in the application of the System Design Factors. It assists the engineer/customer to specify what goal/criteria he wanted to measure and allows the template to be attached or probed onto a subsystem, a component, an object, or the whole system itself just like in the previous examples. This provides the metrification mechanism to quantify the various aspects of design.

1.	Name:	Reliability of Beam Former		
2.	Type:	Probability		
3.	Range:	0.0 to 1.0		
4.	Units:	Units of Probability		
5.	Methods/Principle:	Fault Tolerance, Highly Reliable Component		
6.	Rationale:	Life Critical Function		
7.	Relationship:	Availability, Fault Tolerance, ...		
8.	a. Relational Expression	Positive Correlation, Negative Correlation		
	Quantification			
	a. Type			
	b. Formula	$R(t) = 1 - F(t)$	Actual	
		.989 entered	Required	
		1.01 * Required	Budgeted	
9.	Consistency Rule			
	a. By Aggregation	Use Rule X and Y; Rule X: The probability of the component in series is the product of its probabilities. Rule Y: The probability of the component in parallel use one of voting, voting scheme.		
	b. By Type			
	c. By Design Factor			
	d. By View			
	e. By component			
10.	Reference:	Author's name		
11.	Definition:	Text Book		
12.	Annotation:	Comments		
13.	Next Template:			

FIGURE 4. SYSTEM DESIGN FACTORS TEMPLATE

The initial template was formulated and an example is given in Figure 4 to get the touch and feel of the template. Currently, there are thirteen items in this template. The Name item is a slot holder for the name of a specific design factor (e.g., Reliability of Beam Former). The Type item is a slot holder for the classification of the factor (e.g., Probability). The Range item is a slot holder for the minimum and maximum value or the cardinality of the factor (e.g., 0.0 to 1.0). The Units item is a slot holder for the unit of measurement of the factor (e.g., Units of Probability). The Methods/Principle item is a slot holder for the approaches or techniques that the designer/customer considered to be associated with this factor (e.g., Fault Tolerance, Highly Reliable Component). The Rationale item is a slot holder for

the reason that this factor applies to a specific component/object (e.g., Life Critical Function). The Relationship Item is the slot holder for the list of closely associated factors (e.g., Availability, Fault Tolerance). The Relational Expression field in this item provides the slot for the list of correlations between this factor and its closely associated factors (e.g., Positive correlation, Negative Correlation). The Quantification Item contains the Type and Formula fields. The Type field in this item is the slot holder for either integer, float, double, short, or long. The Formula field in this item currently provides the slot for three mathematical expressions. They are (1) actually calculated (e.g., $R(t) = 1 - F(t)$), (2) required to be a specific value (e.g., 0.989), and (3) budgeted by designer or customer (e.g., $1.01 * 0.989$). The Consistency Rule Item consists of By-Aggregation, By-Type, By-Design Factors, By-View, and By-Component rules. For example, By-Aggregation field provides a slot that holds the rule for governing this factor consistency through out the hierarchy (e.g., Use Rule X and Rule Y). The Reference item is a slot holder for the source of reference or the name of the author that this factor has been formulated by. The Definition item is a slot holder for the clarity for this factor. The Annotation Item is the slot holder for commenting relevant information or providing warnings related to this factor. Lastly, the Next Template item is not completely defined at this time, but it is the slot holder for any detail specification that may not require the customer's direction.

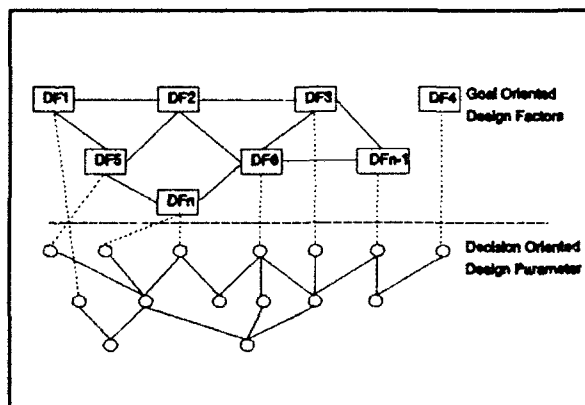


FIGURE 5. DESIGN FACTORS DEPENDENCIES GRAPH

The advantage of this template is not just to ease the use of the factors but it also allows the designer/customer to take the available factors and customize his own design factors that are appropriate for his specific needs. It is up to the Engineer/customer to decide the important and

unimportant factors and formulate the design goal and design decision that the end-result system must meet. The overall design goal and design decision of the system can be described by the System Design Factors dependencies graph shown in Figure 5.

The upper half of the graph is referred to as the goal oriented design factors, while the lower half is referred to as the decision oriented design parameters. The goal oriented is independent of implementation model while the decision oriented is dependent on the implementation model. It would be ideal for the design to be implementation independent in design phase, however in practice it is not always the case. SDF dependencies provide the linkage between the implementation independent (Design Goal) and implementation dependent (Design Parameter). The SDF dependencies graph assists the engineer/customer in understanding the behavior change of the individual factor. These changes are based on its' closely and loosely associated factors. The behavior of each subsystem, component, object, or the whole system with respect to different factors (design goals) can be analyzed separately or simultaneously.

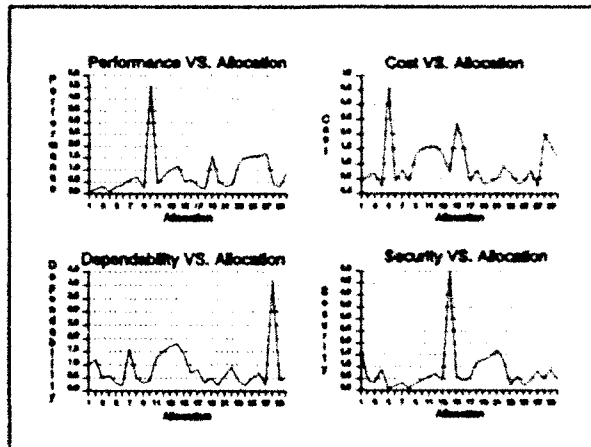


FIGURE 6. SINGLE CRITERIA OBJECTIVE FUNCTION

Although the scope of this paper is not to cover Design Structuring and Allocation Optimization methodology, it is worth showing some applications of SDF with such a method [HHN90a], [HHN90b], [HHN91], [HHN92]. Assume that a customer procured a contract to develop a system such that the end-result system is required to meet certain measurements in terms of Performance, Dependability, Cost, and Security. As illustrated with the previous template example, the engineer can specify and attach these required factors to the design. Based on the design goal and design

parameter, the engineer can tailor the single criteria or multi-criteria objective function for optimization [NaF91].

This design is then optimized based on the tailored objective function. The first approach that the engineer could take is to optimize the design with a single criteria objective function (shown in Figure 6) and then overlay the result (shown in Figure 7) to execute trade-off analysis [Dos91]. The second approach is to optimize the design with multi-criteria objective function (shown in Figure 8). The first approach optimizes the criteria one at a time, while the later approach optimizes these criteria simultaneously.

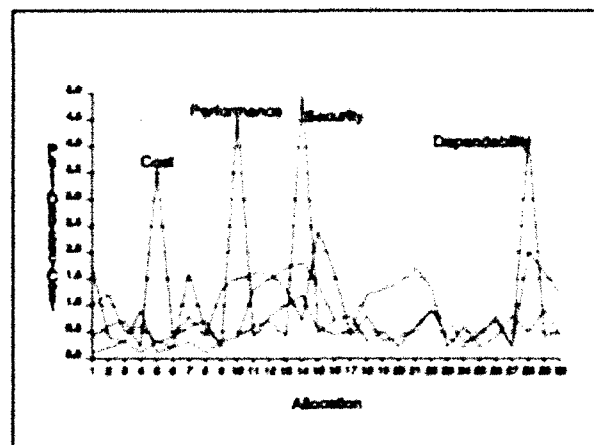


FIGURE 7. SINGLE CRITERIA OBJECTIVE FUNCTION OVERLAY

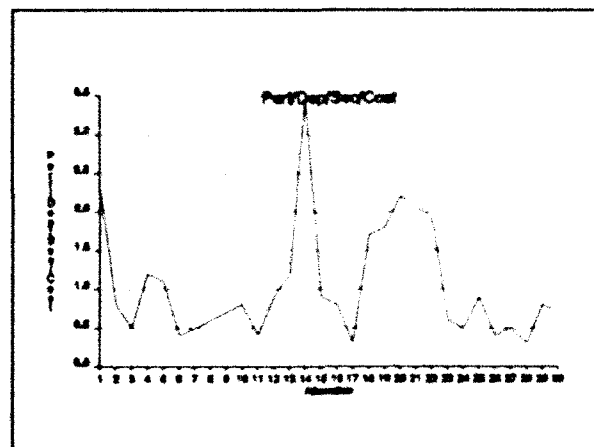


FIGURE 8. MULTI-CRITERIA OBJECTIVE FUNCTION

The results of single and multi-criteria objective function together with the SDF dependencies graph provide the engineer with a better understanding of the system under design. By understanding the physical nature or correlation

among the factors, the designer/customer can predict the behavior and performs effective trade-off. The application of the SDF with optimization here merely demonstrates some utility of the SDF. SDF can be applied through-out various phases of system engineering. It is a critical component in system engineering.

5 SUMMARY OF CURRENT STATUS

A list of System Design Factors was generated and structured in the taxonomy format. There are eleven main groupings of factors and their closely associated factors defined so far. The relationship of these factors is not well understood at the present time but we are attempting to correlate these factors as this effort progresses. An initial System Design Factors technical report is drafted. This draft provides a detailed description of each design factor. The description consists of the terminology, definition, source, metrics, example, usage, and note. The terminology provides the commonly used vocabulary word. The definition provides the meanings of the factor. The source provides the reference of the definitions. The metrics [JuA91] provide the unit of measurement (dimension) of the factor. The example provides some illustration of the factor. The usage provides the cases when, where, how, and why to apply the factor. Lastly, the note provides any relevant information or warning related to the factor. Initial SDF template and example were demonstrated to get the feel of the formulation. The prototyping of the SDF template is underway. Initial System Design Factors focus group has been established to collaborate and to clarify issues in the SDF formulation.

6 CONCLUSION AND FUTURE PLAN

The goal of this effort is to generate a list of System Design Factors. These factors are intended to be used throughout the whole system engineering process. For instance, they are used to specify in the requirements phase, encapsulate in the capturing phase, quantify and evaluate in the analysis phase, characterize in the optimization phase and, justify in the design trade-off phase. These factors are critical to the system engineering process.

The Future Plans included refining, restructuring and streamlining (if necessary) the System Design Factors. More dedicated research

effort is being considered to focus on a smaller but widely use set of design factors. From this smaller set of design factors, intensive correlation will be studied. The formulation will be incorporated into the sonar example [Hoa91] and the Destination Level I Prototype [HNH92], [HNH91] in other research efforts for testing and refining.

The lessons learned in this effort will benefit the whole systems engineering community. The list is expected to evolve as this effort progresses. This is a collaborative effort among Naval Surface Warfare Center (NSWCDDWODET), DoD, other government agencies, Industry, and University communities.

ACKNOWLEDGEMENT

The authors would like to thank Ngocdung T. Hoang, My-Hanh N. Trinh, Charles Whelan Jr., Eric Ogata, Dong Choi, Michael Jenkins, Michael Edwards, Paul Wallenberger, Kate Murphy, Tamra Moore, Dr. Ed Cohen, Dr. William Farr and Dr. Harry Crisp of Naval Surface Warfare Center; Dr. Carl Schmiedekamp of Naval Air Warfare Center; Evan Lock of Computer Command and Control Company; Nick Karangelen of Trident System Incorporated; Robert Goettge of Advance System Technology; Dr. Jane Liu and Dr. Kwei-Jay Lin of University of Illinois Urbana Champaign; Dr. Kishor Trivedi of Duke University; Geoffry Frank of Research Triangle Institute; Dr. Insup Lee of University of Pennsylvania; Dr. Osman Balci; Dr. James Arthur, and Dr. Richard Nance of Virginia Polytechnic Institute and State University and everyone involved in this effort.

REFERENCE

- [BIF90] Blanchard and Fabrycky, Systems Engineering and Analysis, 1990
- [BoB85] Bowen, B. A. Brown, W. R., System Design: Volume II of System Design for Digital Signal Processing, Prentice-Hall, Inc., 1985
- [CYH91] Choi, D. Youngblood, J. Hwang, P., "Modeling Technology for Dynamic Systems", Proc. 1991 Systems Evaluation and Assessment Technology Workshop, Aug 1991.
- [Dos91] Daskocil, D., "Modeling Techniques to Support Diagnostics System Trade-off", Proc. 1991

Systems Evaluation and Assessment Technology Workshop, Aug 1991.

[EdH91] Edwards, M. and Howell, S., "Requirements Specification and Traceability (RESPECT): A Requirements Methods for Large Complex Systems", Proc. 1991 Systems Design Synthesis Technology Workshop, Sep 1991.

[HNH92] Howell, S. Nguyen, C. Hwang, P., "Design Structuring and Allocation Optimization (DeStinAtiOn): A Front-end Methodology for Prototyping Large, Complex, Real-Time Systems", Proc. Hawaii International Conference on System Sciences, IEEE Computer Society Press, Los Alamitos, CA, Jan 1992, Vol. II, pp 517-528.

[HNH91] Howell, S. Nguyen, C. Hwang, P., "System Design Structuring and Allocation Optimization (DeStinAtiOn)", Proc. 1991 Systems Design Synthesis Technology Workshop, Sep 1991.

[HHN90a] Howell, S. Hwang, P. Nguyen, C., "Expert Design Advisor." Proc. 5th Jerusalem Conference on Information Technology (JCIT), IEEE Computer Society Press, Los Alamitos, CA, Oct 1990, pp 743-756.

[HHN90b] Howell, S. Hwang, P. Nguyen, C., "Expert Design Advisor." Naval Surface Warfare Center Technical Report, TR-90-46, Oct 1990.

[Hoa91] Hoang, ND. T., "Essential Views of Systems Development", Proc. 1991 Systems Design Synthesis Technology Workshop, Sep 1991.

[JeY91] Jenkins, M. and Yeh, C., "An Approach to Design of Processor Networks Based On Massively Interconnected Models", Proc. 1991 Systems Design Synthesis Technology Workshop, Sep 1991.

[Joh85] B. W. Johnson, Design and Analysis of Fault Tolerant Digital Systems, Addison-Wesley Publishing Company, 1985.

[JuA91] Juttelsatd, D. and Arnold, C., "The Next Generation Computer Resources (NGCR) Operating System Metrics Project", Proc. 1991 Systems Evaluation and Assessment Technology Workshop, Aug 1991.

[Kam91] Kamat, V., "Computer System Evaluation: Paths and Pitfalls", Proc. 1991 Systems Evaluation and Assessment Technology Workshop, Aug 1991.

[NaF91] Mansour N., and Fox, G., "Physical Optimization Methods for Allocating Data to Multicomputer Nodes", Proc. 1991 Systems Design Synthesis Technology Workshop, Sep 1991.

[SvL76] Svobodova, and Liba, Computer Performance Measurement and Evaluation Methods: Analysis and Applications, 1976

[WaH91] Wallenberger, P. and Howell, S., "Real-Time Dependable Systems Design", Proc. 1991 Systems Design Synthesis Technology Workshop, Sep 1991.

A Method for the Assessment of System Designs

John Litke

**Grumman Corporate Research Center MS.
A08-35**

Bethpage, New York 11714-3580

Abstract

Maturing software CASE tools and more defined software processes now provide the foundation for changing software production from an art into an engineering discipline. An essential next step is to provide objective means to assess the intrinsic qualities of software. In particular, intrinsic properties like complexity have been shown to dominate the development and maintenance cost drivers. This paper reports on a method to objectively assess software qualities related to complexity, including modularity, cohesion, maintainability, etc. The new methods are described and successful application to large DoD software systems reviewed. The technique should be extendable to assess specific structural properties of software related to security, safety, and reliability.

Introduction

System design assessment can have many goals, of which the most common are risk assessment or requirements verification. One reason assessment activity is required is that software as-built is almost never compliant with the program intent. One important cause of non-compliance in software systems is that software is usually more complex than necessary and this unnecessary complexity increases risk and raises the development, verification, and validation costs.

The adverse effect of complexity on the software life cycle cost is not just intuitive, there is documented evidence of its effect. In particular, it is known to be a dominant cost driver for both the development and maintenance phases, to increase the cost of test and integration, to make the

achievement of properties like security very difficult, and to make maintenance expensive. For example, the COCOMO costing model provides cost multipliers that help determine the cost of a software system [1]. Figure 1 shows the ratio of the maximum to the minimum of each cost driver multiplier to illustrate how sensitive the total cost function is to a reduction in the value of a cost driver. Complexity is the cost driver with the most leverage. That is, a proportionate reduction in the system complexity affects the system cost more than a proportionate reduction in any other cost driver. Complexity reduction has even more economic leverage than the use of modern software practices or software development tools!

In addition to cost, complexity has adverse effects on many system properties such as security, safety, and fault tolerance. These attributes need high assurance, and unnecessary complexity in the system makes such assurance either very costly or impossible to obtain. Further, the life cycle cost of software is critically affected by our ability to easily maintain and modify software, and many studies have shown that overly complex systems have an inordinate maintenance cost [2,3].

Despite the advantages of reducing software complexity, no system can be arbitrarily simple: requirements impose a minimal complexity upon any solution. Instead, we intend to manage and control complexity growth and, if possible, to reduce inessential complexity in software systems.

To do this, we are developing an objective software assessment technique to characterize software complexity, focusing on those complexity issues that most likely affect life cycle cost by affecting software modularity, reusability, maintainability, etc. We have begun by assessing software implementations for complexity properties, but we believe that our technique can be extended to assess the complexity of software designs and maybe even the complexity of software requirements.

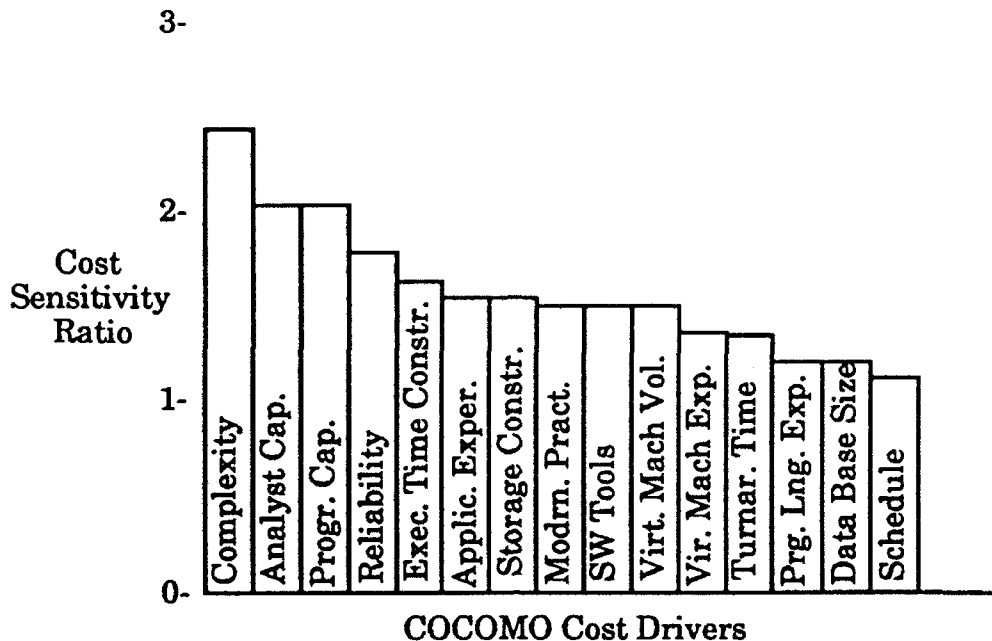


Fig. 1: Cost Driver Sensitivity

By itself, this goal is not sufficient. To ensure that the assessment technique is useful, we have:

- Used it to determine the complexity in software produced by good practice,
- Developed techniques that reduce complexity in reasonable instances and proved those techniques work,
- Assessed the benefits of the entire process.

Although our initial results reported here are preliminary, we believe there is sufficient promise that we have committed to the construction of a second generation assessment technology.

Technical Approach

Our approach to assessing software complexity is to apply pattern analysis technology to software systems. We believe that as software systems are made more regular (less complex), their design integrity improves and the life cycle cost should decrease. Generally, pattern analysis as a conceptual approach has the

advantage that it looks for high level characteristics, assessing large scale, system level attributes that more traditional, detail-oriented assessments miss. On the other hand, pattern analysis as an approach is often confused by small variations and either fails to recognize good patterns, or identifies good patterns with slight variations as bad. This sensitivity to false identifications is well known and is one of our more significant challenges.

The potential applicability of pattern analysis for software complexity assessment is relatively easy to describe, even as the real utility is less obvious. For example, it is commonly accepted that well-engineered software is modular, hides information within modules, and minimizes the complexity of inter-module couplings. These characteristics, accepted since the 1960's, are the foundation of the work of authors such as Myers and Parnas. One difficulty with the concepts is that they are relative and descriptive, rather than absolute and prescriptive. Although they suggest that modules should be more tightly coupled to their internal elements than to external elements, the definition of "more"

or the specification of how much external coupling is acceptable is answerable only in the context of specific software instances. Nevertheless, a knowledgeable software engineer can sometimes examine the modularity properties of a program and quickly assess whether or not it is acceptable, even though determining specific modification suggestions may be much more time consuming. Making such an assessment process more objective is one goal of our pattern analysis technique.

Consider a program as a complex network of relationships such as data references, control references, semantic dependencies (e.g., types), exception propagations, etc. If the program was constructed with definite methods, we should expect that the methods or their goals are evidenced in the final product. In particular, because most software engineering methods emphasize the control of relationships (e.g., interfaces, data references), an analysis of relationships should reveal the effect of the software design and construction disciplines. However, the analysis of relationships is challenging because appropriate relationships are not regular constructs like bricks. If they were regular, the detection of inappropriate patterns would be as easy as detecting the irregularities in a brick wall. Instead, the net effect of the many indirect couplings between parts of a software system provide a much more complex picture that is difficult to understand.

The complex network of interconnections within a program defines the graph to be analyzed. We will use the analogy of analyzing a business organization for comparison. The usual first step is to focus on the small details of the organization, the specific work rules, time card practices, personnel policy, etc. Although such information can characterize the spirit and general tone of an organization, it does relatively little to clarify its organizational structure. By analogy, analyzing software for density of comments, use of risky constructs, etc. gives only a limited and sometimes misleading picture of the overall system.

At the next level of analysis, we would identify the minor organization structures (e.g., work groups within departments) and their relationship to each other. The minor structures often do not respect formal organizational boundaries but represent teams of organizationally dispersed workers that cooperate extensively. In the same way, we can identify program elements (e.g., functions, subprograms, tasks in Ada) and ask how they cooperate within the program to achieve specific tasks.

At a still higher level, an organization's minor elements form major working groups. These larger groups characterize the real organizational points of control and critical workgroups. In the same way, the smaller elements of a program form subsystems or abstractions that characterize the larger compositional structure of the program.

This process of identifying smaller structures and grouping them into larger structures in a recursive manner may be repeated many times, from different points of view, to properly characterize a software system or an organization. For example, if we analyzed the organizational flow of forms, or of capital authorizations, or of proposal writing, or of the coordination of social gatherings, we might obtain very different pictures. Similarly, we expect that the analysis of data references, type references, calling references, exception processing, etc. will reveal different structures in a software system.

Organizations, like software, are not perfect; they do not operate exactly as they were designed to do. Therefore, we should expect to find work groups of all sizes that differ significantly from the formal organization chart. When this is so, it is essential to determine whether or not the organization (or the software system) should be reorganized to better match the work processes. Conversely, we might also find that work relationships contain long chains of essentially non-functional relationships. These are obvious candidates for simplification in both software systems and organizations.

In brief, our analysis approach examines the patterns of relationships within a software system to discover the naturally cohesive elements within it. We do this by assuming that a collection of modules can be usefully described as a cohesive group if its members more complexly relate to other group members than they relate to the rest of the system. From another perspective, a cohesive group is perceived by a relatively low complexity boundary between it and the rest of the system. Our approach does not criticize a grouping by labeling it an inappropriate solution to the design problem, but rather it identifies unusual complexity in a solution. In this sense, our approach criticizes not the system design itself, but rather the workmanship of decomposing the design into elements.

The utility of this approach depends upon the efficacy of cluster analysis algorithms. Cluster analysis performance depends upon our choice of measures, metrics for coupling strength, and upon our computational approach. A summary of some considerations for algorithm design is in Table 1. (More information on cluster analysis can be found in reviews such as [12].)

The table lists properties (such as a measure) and typical examples for the software analysis problem, e.g., data references. Then it suggests a probable strength of this measure and a probable weakness. For data references, we expect that they should be well controlled by traditional design methods and that therefore patterns defined by data references should not conflict with the design. A potential weakness of this measure is that, if all methods do succeed in controlling coupling by data references, then the violations to be revealed will be few and the insight gained slight. The coupling strength criterion property describes the method for defining coupling to be present. The simplest approach is to define a metric on a specified measure and then to say that any coupling stronger than a specific metric value is considered coupled. More adaptive

approaches use fractions of local or global averages of the metric to define coupling. Finally, algorithms can be divisive or agglomerative. Divisive methods tend to be computationally expensive, but are more able to discover imperfect clusters. Agglomerative methods, on the other hand, are usually inexpensive and identify small structures more readily, but they can also be easily misled by the presence of incidental couplings or unusual structures

Our initial attempt used an agglomerative method using data and call references, and an absolute coupling criterion. From this experience, and using Table 1 as a table of expectations (not results), we chose one agglomerative method and one divisive method for additional evaluation as suggested in Table 2. As complexity is identified, we also will explore methods that help reduce or control complexity as summarized in Table 2.

Complexity Identification

The method of decoupled groups identifies groups of program elements that are decoupled by a specified criteria, such as data coupling, using an bottom-up, agglomerative clustering technique. We began with an absolute measure of coupling, e.g., any software module that references data declared in another module is coupled to the module that contains the data. This approach is related to the ideas of Parnas and Myers. Although the technique did, in fact, provide useful insight, we also found that it was too "ideal" an approach and often provided unreasonable criticism of reasonable software structures. Because many programs appear to have clusters of modules that relatively intensively reference each other's visible data, only one reference from one cluster to another ties both clusters together. In this way, many otherwise reasonably structured programs appear to be totally coupled, whereas the removal of only 6-8 isolated data references would reveal a more reasonable modularization of 8-10 sub-systems. However, the approach is helpful for

Property	Example	Strength	Weakness
Measure	Data references	Often corresponds to method goals	Programs are often well designed
	Calls	Reflects functional decompositions	Extensive coupling to primitives
	Types	May reflect design abstractions	Unexplored May be uncontrolled
	Withs	Includes previous three measures	Insufficient for corrective action
	Exceptions	Reliability emphasis	Unexplored
Coupling Strength Criterion	Absolute	Cheap, easy	Too sensitive
	Local average	Adapts to sub-system structures	False alarms Splintering
	Global average	Adaptable	Poor for inhomogeneous systems
Algorithm	Top-down, divisive	Can reveal high level structures more reliably	Often computationally costly
	Bottom-up, agglomerative	Often computationally inexpensive More likely to reveal small reusable modules	Very sensitive to measure choice

Table 1: Properties of Cluster Analysis Algorithms

security and safety analyses where rigid criteria are necessary to provide high assurance. Our second generation technology will use all five measures in Table 1 to more fully explore the potential of this approach. We will explore the use of a coupling strength criterion that uses the average coupling of the group gathered so far. From experience in other venues [9], this method works well when there are clear and easily discriminable clusters in the relationships.

A complementary approach identifies sub-systems within the larger system that are locally more coupled or cohesive within the

group and thereby less complexly related to the rest of the system. Instead of looking for sub-systems with little or no coupling to other sub-systems, we look for graph partitions with maximal internal cohesion or integrity. This difference is an essential point.

Previous researchers, including ourselves, looked for sub-systems that were coupled to the rest of the system by a specific strength or less, that is, the focus is on the relative pairwise coupling strength. We now believe that it is as important to use inverse logic -- look for sub-systems that are inherently more coupled to themselves than to the rest of the system. This is a much harder but, we

Goal	Method	Strength	Weakness
Complexity Identification	Decoupled groups	Corresponds to method goals	Few programs are good enough
	Maximally coupled groups	Works on any program	May not map easily to design or method goals Algorithms are difficult
Complexity Reduction	Isolated Corrections	Cheap, easy, effective	May not provide essential simplification
	Sets of changes	Supports cost/benefit analyses	Algorithms are difficult

Table 2: Pattern Analysis Methods for Managing Complexity

believe, a more promising technique. We have designed a top-down, divisive method that uses local average coupling strength and any of the five measures of coupling from Table 1. This method tries to find the lowest complexity boundaries with respect to the complexity of the groups formed thus far, rather than with respect to a context-insensitive definition of low complexity. It will use variations of the Kernighan and Lin [10] clustering methods.

Complexity Reduction

The other two methods in Table 2 are the complexity reducing analyses to identify couplings that can easily be modified or removed in an attempt to reduce complexity. This requires specific information about Ada language structures, and one illustrative result will be described later. However, once these methods remove the clutter, there remains some essential complexity that may take much more effort to remove. We have some proposed algorithms to suggest sets of changes and guide decision making. These algorithms will be implemented and tested in our second generation analysis tool set.

Our approach is unique, but not novel (Table 3). It is related to methods described in the literature for some time, but no one, to our knowledge, has developed a broadly applicable, high level analysis technique. The traditional statement level, detailed

methods, such as Halstead and McCabe, provide statistical characterizations, but little help for modification or repair strategies.

Our approach instead builds upon the module level definition of complexity such as that pioneered by Parnas and Myers. It is our goal to use this sense of inter-module complexity to characterize whole systems so we can identify locally complex subsystems within that whole that should be examined more closely. Yau, Belady, and Card provide early examples of applications of this concept which we have extended and modified to be practical for large Ada systems.

To do these analyses requires an extensive automated analysis capabilities to collect the information from large Ada programs, trace the desired relationships, and identify the complexity relationships of interests. Figure 2 illustrates the architecture of our second generation analysis technology (known as ADAPT) now under development.

We ensure that the ADAPT analysis system will analyze large Ada codes by using a Diana-based tool from the STARS program for semantic analysis of the full Ada language. This tool extracts the relevant information and stores it in a relational database. Many simple questions can be answered by relational queries, but the structural, pattern related questions require

further analysis. We found that exception raising and propagating relationships require more complex infrastructure than

other relationships, but otherwise the technology is similar.

Method	Level	Approach	Strength	Weakness
Grumman Pattern Analysis	System	Multiple	Adaptive	Less mature
Design Stability (Yau [4])	System	Modifiability	Sub-system focus	Single attribute
Data Binding (Belady [5])	System	Maintainability		
Design Complexity (Card [6])	System	Design Assessment		
Information Hiding (Parnas [7])	Module	Complexity Control	Well understood	Local evaluations
Coupling (Myers [8])	Module	Understandability		
Adamat/Logiscope	Detail	Multiple Statistical Characterizations	Mgmt. Tool	Limited applications
McCabe/Halstead	Detail	Statistical Characterization	Code Sensitive	Context insensitive

Table 3: Comparison of Complexity Characterization Methods

When we analyze software, we produce not only insights into the general complexity of the software and information on what groups are coupled to others, but the analysis also provides many helpful listings as a by-product, such as variables that can be constants, unnecessary *with* statements, etc. For example, the analysis of the exception handling methods of one 20,000 line system provided detailed tables of information, of which Table 4 is an extract. The capability for a full static trace of exception propagations will be available in our new technology, but even limited statistics like this can be helpful. For example, this program has 69 *when others*, which seems rather high. Further, four of its *others* handlers are null, as are 18 of the named exception handlers. This information

suggests that the exception handling logic might not be fully developed. The complete result of a typical analysis is about 100 pages of information. Although, for research purposes, we usually analyze a system to see whatever can be seen, we recognize that a real user would have specific applications in mind and would require a much more focused analysis. Examples of such role differences are illustrated in Table 5.

To address these different roles, we have taken care that all grouping algorithms retain the linkages from the low level details that caused the groupings to the high level groups that result. This permits us to support the more detailed requirements of the verifier or maintainer that need specific information on cause and effect.

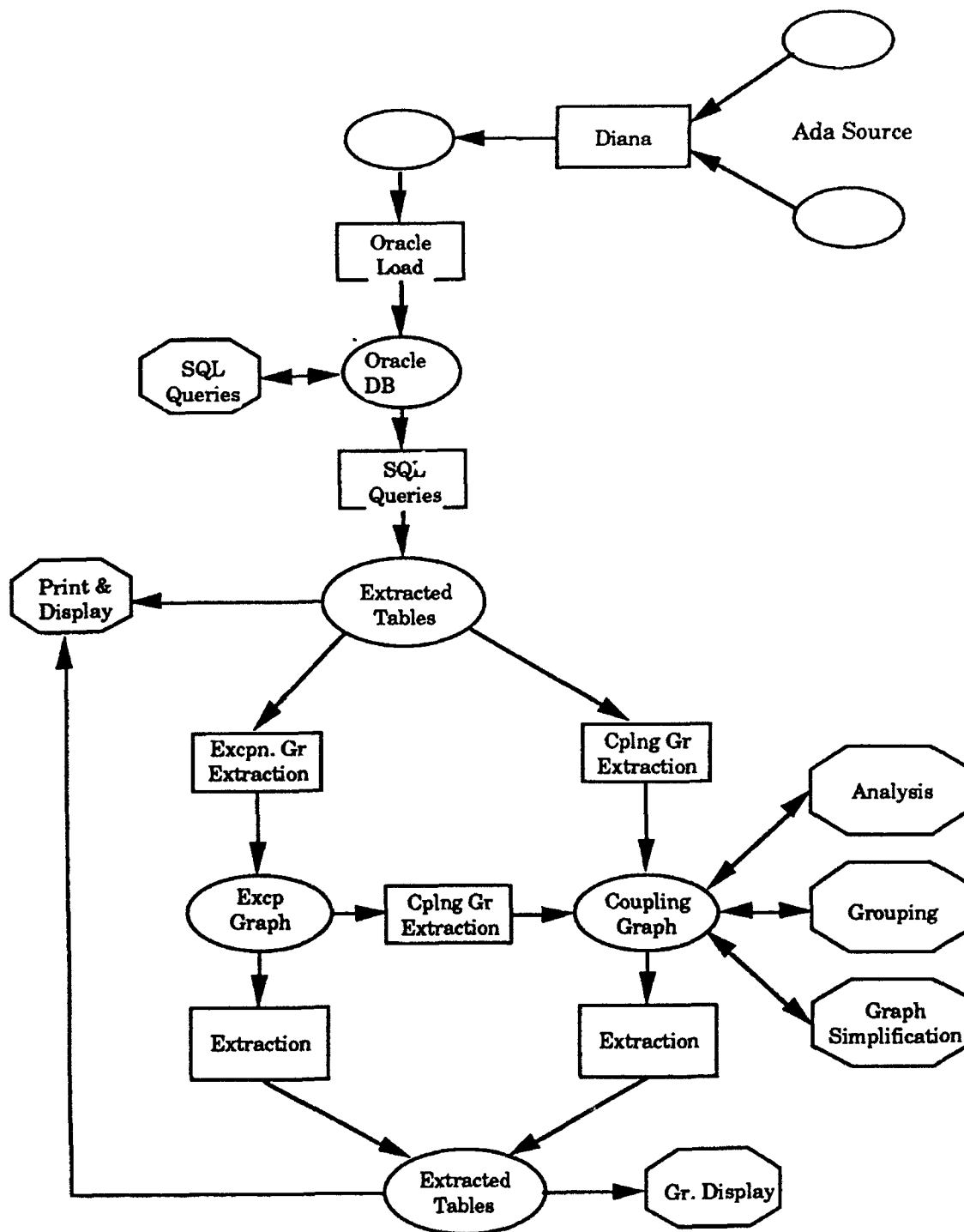


Fig. 2. ADAPT Analysis Tool Structure

Attribute	Count	Raise Stmt	Specific Handlers
Pre-defined exceptions explicitly raised	2	4	63
User-defined exceptions raised	29	80	145
User exceptions defined but not used	1		

Table 4: Example of exception information

User Role	Possible Goal	Application
Program Office	Software Architecture Assessment	Risk Assessment Reuse Potential
Engineering Management	Design Evaluation	Complexity Modularity Maintainability
Verifier	Property Certification	Security Safety Fault Tolerance
Maintainer	Re-engineering Analysis	Cost/Benefit Evaluation Identification of sub-systems

Table 5: Potential Applications

Results

A sample of the Ada software analyses we have done so far is in Table 6.

In all cases, we found that the patterns that were perceivable by the data references criteria, bottom-up, decoupled grouping method were generally congruent with the software designer's intent. On the other hand, we found many examples in reputedly well constructed code where pervasive inter-module coupling was apparently inadvertently introduced during the software construction process. Deciding which couplings could be most profitably removed was not easily accomplished with our first generation tool.

Some code systems (e.g., CAIS) were specifically designed with object-oriented paradigms, while others were designed with

functional abstractions. Although we saw significant differences in the interconnection patterns, there were also significant similarities. This suggests that some pattern analysis techniques are insensitive to the particular design and development paradigm.

In five of these eight examples, we presented our analysis results to representatives of the original implementation team. These conversations confirmed that the insights provided by the techniques are useful to the developers and suggested many ideas for potentially useful patterns. Note that our technology does not produce patterns or structures, but instead reveals structures created during software design and development. In one example, the user was interested in assessing the reuse potential of a software system. Our pattern analysis technique suggested many places where

extensive data coupling made reuse difficult. It is significant that the user did not say "Find all places where data coupling is a problem," but rather the analysis technique directed attention to this system aspect.

In another example, several related code systems were re-engineered to increase the number of shared modules. We applied our analysis technology to before and after versions of the re-engineered software. Although the number of common modules in the re-engineered systems increased by about 65%, our analysis also showed that the number of inter-module references (*with* statements) increased by 153%. This suggested (for this particular experiment) that the benefits of increased commonality were offset by a significant increase in the code system complexity. Although our technique cannot decide whether this is a favorable or unfavorable result, it can reveal the unintended effects of the re-engineering effort.

Another frequent observation was unnecessary complexity in many large software systems that were developed under alleged rigorous methods and 2167A documentation requirements. For example, Figure 3 is a graph of *with* relationships in a 45,000 line program. In this example, many *with* relationships have the pattern illustrated in Figure 4. Here, A *withs* B and C, and B also *withs* C. In some cases, A's direct reference to C is a violation of the abstraction provided by B, and in other cases it is not. However, Ada's *rename* statement permits an easy graph simplification that removes the direct reference from A to C, and instead presents the C interface within the specification of B. This simple change transforms Figure 3 into Figure 5. This does not imply that it is always wise to accept all these transform suggestions, but the magnitude of the potential simplification warrants serious attention to the suggestions.

Name	Type	Lines of Code
GIFRS	Simulation	19,100
TASKIT	Simulation	50,700
SGML	Text Processing	5,400
ADAPT	Program Analysis	45,100
Navy Code	Message Handling.	31,700
CAIS	File System Interface	359,800
DIANA	Language Translation	38,200
SPC Code	Compilers	63,500

Table 6: Analysis Examples

As an unanticipated by-product of our analysis tools, we found several such simplifying transforms that are very helpful to remove some of the more careless complexity raising dependencies. These

include variables that can be constants, variables that should be moved to another package, useless *with* statements, etc. As part of our complexity reducing algorithms for the second generation analysis technology, we will be exploring algorithms

that suggest moving modules from one package to another in an attempt to make Ada package boundaries more closely match the complexity boundaries in the code. This

should help to gradually transform a programmatic representation (i.e., Ada code) to one that more closely matches the

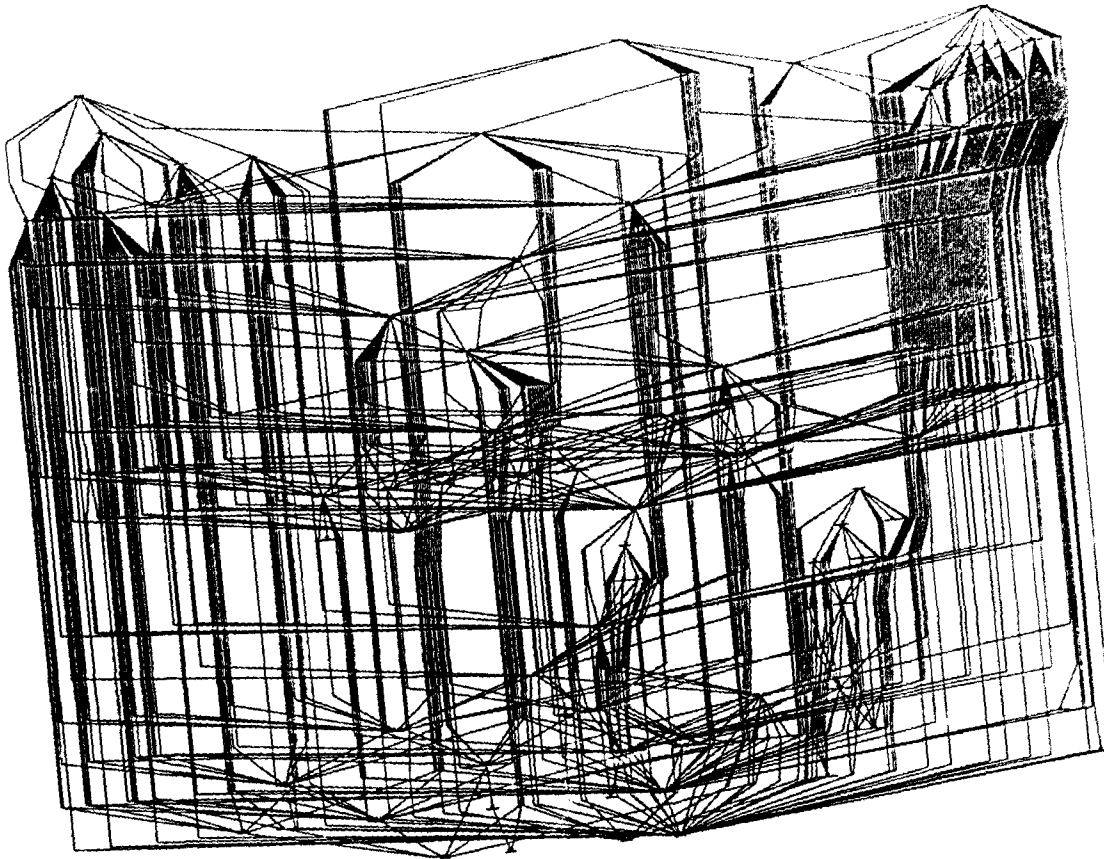


Fig. 3 With relationships

actual structures. (LaMarr [11] recently reported on an interesting case where the disparity between the documentation structure and the actual code structure made understanding very difficult. These transforms should be helpful for reducing that effect of complexity.)

During the analysis tasks, we were limited by our first generation technology to bottom-up, decoupled grouping using several measures of intermodule data references. This was successfully applied to verify checkpointing data sets in a large, distributed Ada application, but in general it has proven inappropriate for many large

programs. As part of the development of new grouping technology, we now can graph relationships between either Ada modules or packages, using constant references, variable references, calls, *with*'s, or type references defining the relationship (and exception reference graphs are under development). Comparing many graphs, it is clear that some large programs are much more complex than others of equivalent size and intent, suggesting that complexity is not inevitably the curse of bigness. We have also noted that data and call references are those relationships that are most likely appropriately structured. For example, Figure 6 is a package-to-package call graph

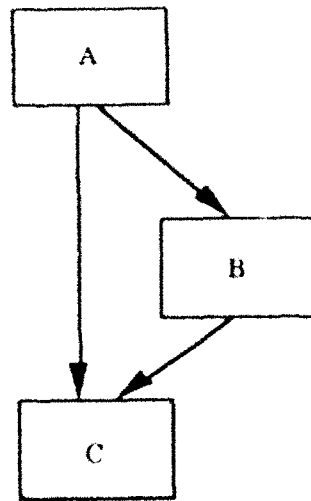


Fig. 4. With reference pattern

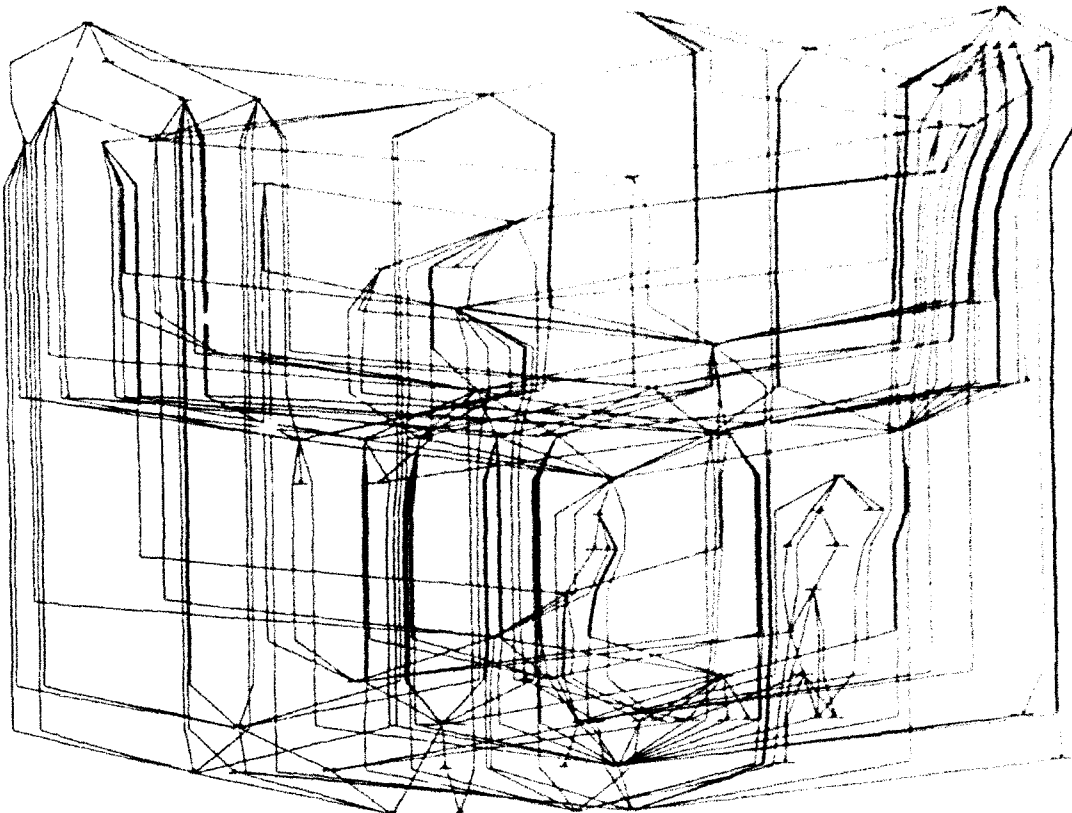


Fig. 5 Simplified with relationships

of a 35,000 line subset of a large military command and control system.

Typically, if we see a reasonably organized call graph such as this, the inter-package data reference graph also is relatively orderly. This is not surprising, because most detailed design methods emphasize reduced data coupling between modules and some form of successive refinement of designs that tends to produce relatively

orderly hierarchical call graphs. In contrast, Figure 7 shows the inter-package type reference graph of the same program. It is much less regular, and more complex. It also has many more packages in it, because many packages contain types but no call references. This result is true for all programs we analyzed so far: that the type reference graph is much more complex than the data or call reference graphs. The only exceptions are a few

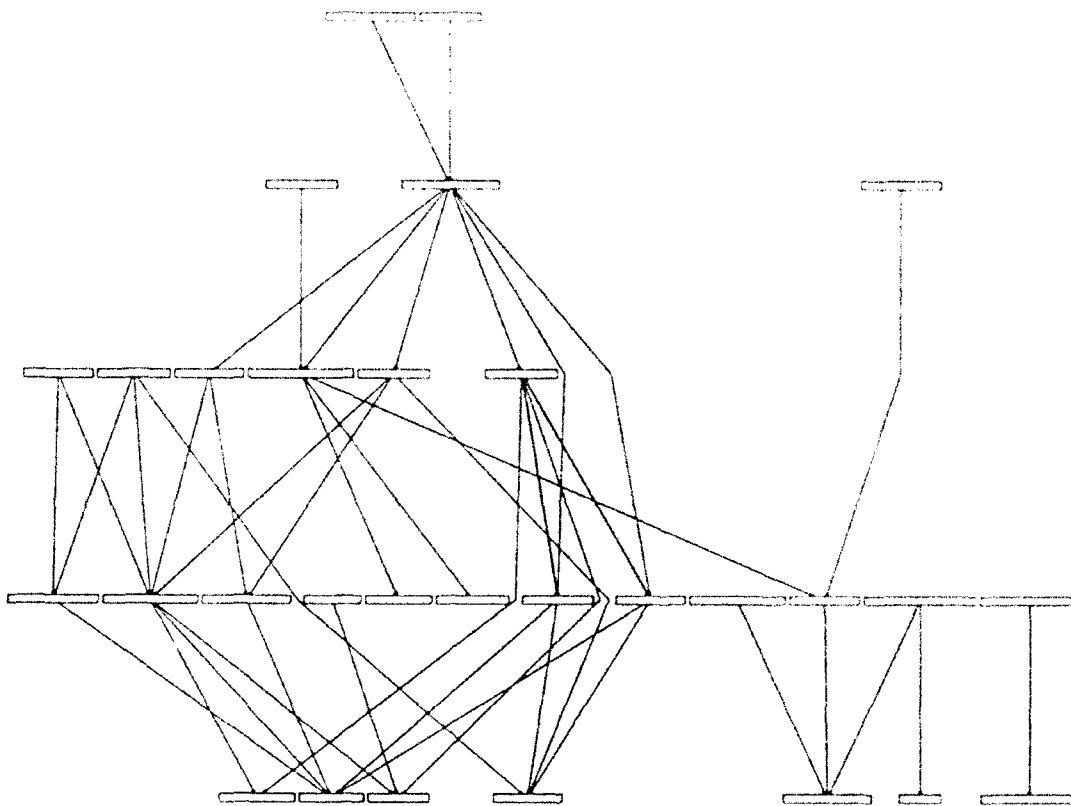


Fig. 6 Package to Package Call Graph

programs whose call and data reference graphs are unusually complex and the type reference graph is equally so, suggesting poor design or implementation problems.

We can not yet suggest movement of type definitions from package to package to determine how much of this additional complexity is superficial and what is

essential. However, these initial observations suggest that, while modern methods can control the complexity of data and call references, they provide much less control of the type definition and reference structure complexity.

Summary

In summary, we believe we have developed an innovative technique to assess the complexity of large software systems and to help reduce or manage that complexity. The pattern analysis technique not only helps identify what is, but also can help guide

modifications toward what ought to be. Although the technique's true strengths and weaknesses are still unknown, we are convinced that there are many effective applications for managing complexity growth or reducing the complexity in software systems.

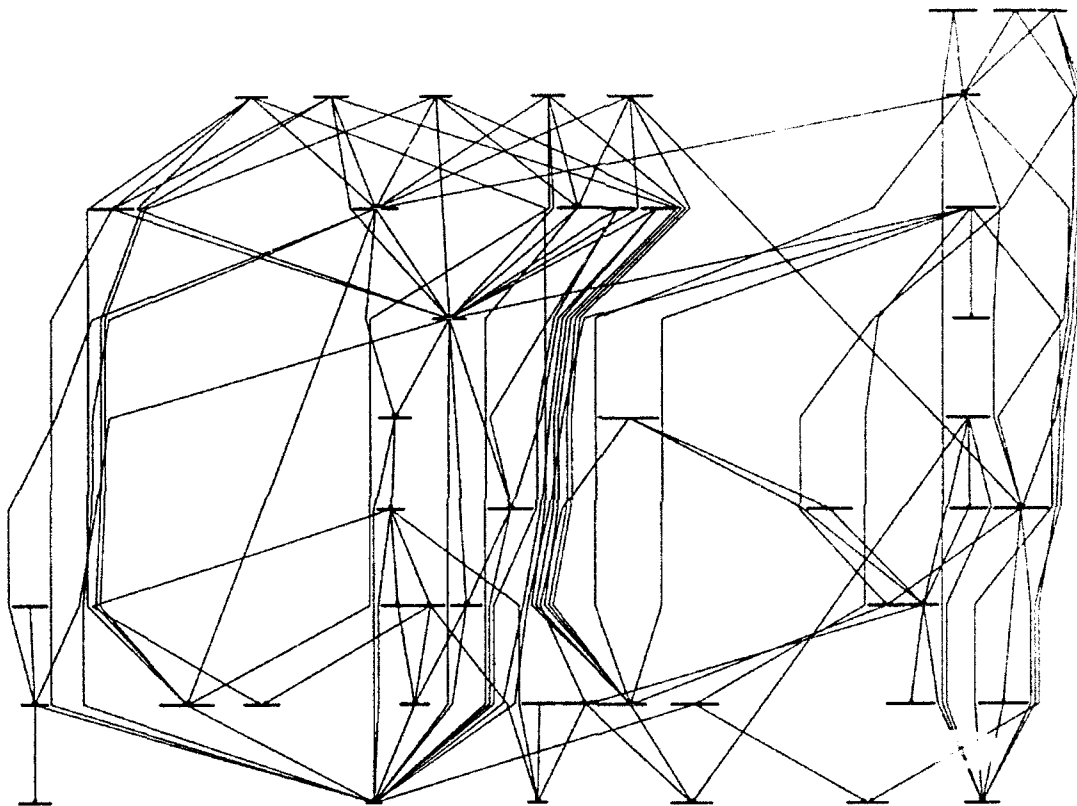


Fig. 7 Package to Package Type Reference Graph

Our focus for now will remain on assessing the quality of systems and helping guide complexity reducing efforts. However, we expect that more advanced applications of the technique should permit us to predict the probable complexity of solutions while still in the design stage. This should help us estimate the life cycle cost of software and should be a major step toward managing the life cycle cost of software, rather than just trying to control it. Even more attractive, but at the horizon for now, is the potential for

evaluating the relative efficacy of proposed software development methods by comparing the relative complexity of their products. This application could be a major step toward engineering new and improved software processes.

Acknowledgements

The extensive software tools required to support this research were designed and constructed by Peter Brennan and Un Fu of the Grumman Data Systems Technology

Department. Their skillful application of the tools to large software systems provided many specific insights summarized in this paper.

References:

1. Boehm, Barry W., "Software Engineering Economics," Prentice-Hall, Englewood Cliffs, NJ, 1981
2. Kafura, Dennis and Geereddy R. Reddy, "The Use of Software Quality Metrics in Software Maintenance," Virginia Polytechnic Institute Technical Report TR-85-33, August 1985
3. Henry, Sallie and Dennis Kafura, "Software Structure Metrics Based on Information Flow," IEEE Trans on Softw. Engr, vol SE-7, no. 5, Sept. 1981, pp 510-518
4. Yau, Stephen S, and James S. Collofello, "Design Stability Measures for Software Maintenance," IEEE Trans Softw. Eng, vol SE-11, no. 9, Sept 1985, pp 849-856
5. Belady, L.A. and C. J. Evangelisti, "System Partitioning and Its Measure," J. of Systems and Software, vol 2, 1981, pp 23-29
6. Card, D.N. and W. W. Agresti, "Measuring Software Design Complexity," J. of Systems and Software, vol 8, 1988, pp 185-187
7. Parnas, David Lorge, Paul C. Clements, and David M. Weiss, "The Modular Structure of Complex Systems," IEEE Trans Softw. Engr, vol SE-11, no. 3, March 1985, pp 259-266
8. Myers, Glenford J, "Composite/Structured Design," Van Nostrand Reinhold, New York, 1978
9. Litke, John D., "A Practical Solution to the Traveling Salesman Problem with Thousands of Nodes," CACM vol. 27, no 12, Dec. 1984, pp 1227-1236.
10. Kernighan, B. W. and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," Bell System Technical J., Feb. 1979, pp. 291-307
11. LaMarr, Yvonne, and William E. Fravel, Jr., "Software Independent Verification and Validation: A Process Perspective," Proc. Tri-Ada 91, pp 408-417
12. Spath, Helmut, "Cluster Analysis Algorithms," Ellis Horwood, Chichester, 1980

Methodology for Validating Software Metrics

Norman F. Schneidewind

Code AS/Ss
Naval Postgraduate School
Monterey, CA 93943

Abstract

We propose a comprehensive metrics validation methodology that has six validity criteria, which support the quality functions assessment, control and prediction, where quality functions are activities conducted by software organizations for the purpose of achieving project quality goals. Six criteria are defined and illustrated: association, consistency, discriminative power, tracking, predictability and repeatability. We show that non-parametric statistical methods like contingency tables play an important role in evaluating metrics against the validity criteria. Examples emphasizing the discriminative power validity criterion are presented. A metrics validation process is defined that integrates quality factors, metrics and quality functions.

Index Terms: metrics validation methodology, metrics validation process, non-parametric statistical methods, quality functions, validity criteria.

INTRODUCTION

We believe that software metrics should be treated as part of an engineering discipline: metrics should be evaluated (validated) to determine whether they measure what they purport to measure prior to using the metrics. Furthermore, if metrics are to be of greatest utility, the validation should be performed in terms of the quality functions (quality assessment, control and prediction) that the metrics are to support.

We propose and illustrate a validation methodology whose adoption, we believe, would provide a rational basis for using metrics. This is a comprehensive metrics methodology that builds on the work of others. These have been validation analyses performed on specific metrics or metric systems for the purpose of satisfying specific research goals. Among these validations are the following: 1) function points as a predictor of work hours across different development sites and sets of data [1]; 2) reliability of metrics data reported by programmers [3]; 3) Halstead operator count for Pascal programs [10]; 4) metric-based classification trees [16]; 5) evaluation of metrics against syntactic complexity properties [17].

Our approach to validation has the following characteristics: 1) The methodology is general and not specific to particular metrics or research objectives. 2) It is developed from the point of view of the metric user (rather than the researcher), who has requirements for assessing, controlling and predicting quality. To illustrate the difference in viewpoint, we can make an analogy with the automobile industry: the manufacturer has an interest in brake lining thickness, as it relates to stopping distance, but from the driver's perspective, the only meaningful metric is stopping distance! 3) It consists of six mathematically defined criteria, each of which is keyed to a quality function, so the user of metrics can understand how a characteristic of a metric, as revealed by validation tests, can be applied to measure software quality. 4) The six criteria are: association, consistency, discriminative power, tracking, predictability and repeatability. 5) It recognizes that a given metric can have multiple uses (e.g., assess,

control and predict quality) and that a given metric can be valid for one use and invalid for another use. 6) It defines a metrics validation process that integrates quality factors, metrics and functions.

The paper is organized as follows: First, a framework is established which pulls together the concepts and definitions of quality factor, quality metric, validated metric, quality function, validity criteria, and a metrics validation process. These concepts and definitions are integrated by the use of a metrics validation process chart. In this section we show how validity criteria support quality functions. Next, we indicate why non-parametric statistical methods are applicable to and compatible with the validity criteria. This is followed by an example of metrics validation, using the discriminative power validity criterion. Lastly, some comments are made about future research directions.

FRAMEWORK

The framework of our metrics methodology consists of the following elements, which are keyed to Figure 1: quality factor, quality metric, validated metric, quality functions, validity criteria, and metrics validation process. In Figure 1, we use the notation [Project, Time, Measurement] to designate the project, time (e.g., life cycle phase) and type of measurement (quality factor, quality metric). We use V to designate the project in which a metric is validated and A to designate the project in which the metric is applied.

This diagram is interpreted as follows:

- o The events and time progression of the validation project are depicted by the top horizontal line and arrow. This time line consists of Project 1 with metric M collection in Phase T1 (step 1); factor F collection in Phase T2 (step 2); and validation of M with respect to F in Phase T2 (step 3).
- o The events and time progression of the application project are depicted by the bottom horizontal line and arrow. This project is later in chronological time than the validation project but has the same phases T1 and T2. This time line consists of Project 2 with metric collection M' in Phase T1 (step 4); application of M' to assess, control, and predict quality in Phase T1 (step 5); collection of factor F' in Phase T2 (step 6); and revalidation of M and M' with respect to F and F' in Phase T2 (step 7).
- o Metric M' is the same metric as M but, in general, it has different values since it is collected in a different project. The same statement applies to F' and F.

Each element is defined and described in greater detail in the following sections.

Quality Factor

A quality factor F (hereafter referred to as "factor" or "F") is an attribute of software that contributes to its quality [13], where

software quality is defined as the degree to which software possesses a desired combination of attributes [14]. For example, reliability (an attribute that contributes to quality) is a factor. A factor can have values, such as the error counts F_1, \dots, F_n in a set of software components (i.e., an element of a software system, such as module, unit, data or document [13]). We define F to be a type of metric that provides a direct measure of software quality [6]. This means that F is an intrinsic indicator of quality as perceived by the user, such as errors in the software that result in failures during operation. We denote F as the factor in V and F' as the factor in A . F and F' are shown as collected at point 2 and at point 6, respectively, in Figure 1.

Quality Metric

A quality metric M (hereafter called "metric" or " M ") is a function (e.g., cyclomatic complexity $M = e - n + 2p$) whose inputs are software data (elementary software measurements, such as number of edges e and number of nodes n in a directed graph) and whose output is a single numerical value M that can be interpreted as the degree to which software possesses a given attribute (cyclomatic complexity) that may affect its quality (e.g., reliability) [15]. For example, if there are two components 1 and 2 with $M_1 = 3$ and $M_2 = 10$, this may indicate that the reliability of 1 may be greater than the reliability of 2. Whether this is the case depends upon whether M is a valid metric (see below). We define M to be an indirect measure of software quality [2,6]. This means that M may be used as a substitute for F , when F is not available, as is the case during the design phase. M is shown as collected at point 1 in Figure 1.

It is important to recognize that, in general, there can be a many-to-many relationship between F and M . For expository purposes we limit our examples to one-to-one or one (F) to many (M) relationships.

Validated Metric

A validated metric is one whose values have been shown to be statistically associated with corresponding factor values (e.g., M_1, \dots, M_n have been statistically associated with F_1, \dots, F_n for a set of software components 1, ..., n) [13]. A validation test of M with respect to F is shown at point 3 in Figure 1. We denote M' as a validated metric. Since M is validated with respect to F , it is necessarily the case that F is valid. Therefore we say that F is valid by definition, as a result of wide acceptance or historical usage (e.g., error count).

Since F is a direct measure of quality, it is preferred to M whenever it is possible to measure F sufficiently early in the life cycle to permit quality to be assessed, controlled and predicted (see below). However, since this is usually not the case, the need for validation arises. We also note that since the cost of finding and correcting errors grows rapidly with the life cycle, it is advantageous to have approximate early (leading) indicators of software quality. (Analogously, one could posit that the Dow Jones stock price average (M) is an approximate leading indicator of Gross National Product (F) in the

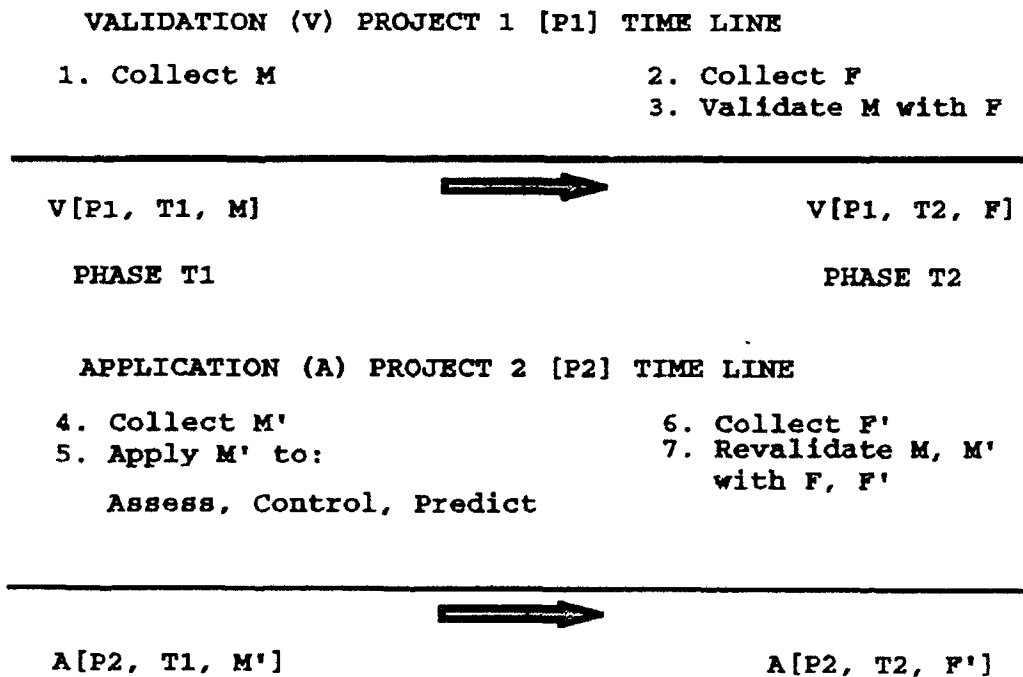


Figure 1. Metrics Validation Process

American economy and conduct a validation test between the two). Thus, we can formulate the following policy with respect to software measurement: When it is feasible to measure and apply F , use it; otherwise, attempt to validate M with respect to F and, if successful, use M' .

Quality Functions

Quality functions are activities conducted by software organizations for the purpose of achieving project quality goals. Both product and process goals are included. The quality functions that are pertinent to this metrics methodology are: assessment, control and prediction.

Quality Assessment

Quality assessment is the evaluation of the relative quality of software components. "Relative quality" is the quality of a given component compared with the quality of other components in the set (e.g., if M' is cyclomatic complexity, the quality of component 1, with $M' = 3$, may be better than the quality of component 2, with $M' = 10$). Validated metrics are used to make a relative comparison of the quality of software components. The purpose of assessment is to provide software managers with a rational basis for assigning priorities for quality improvement and for allocating personnel and computer resources to quality assurance functions. For example, priorities and resources would be assigned on the basis of relative values (or ranks) of M' (i.e., the most resources would be assigned to the components with the highest (lowest) values (or ranks) of M'). M' is shown collected at point 4 in Figure 1 and used for assessment at point 5.

Quality Control

Quality control is the evaluation of software components against predetermined critical values of metrics (i.e., value of M' which is used to identify software which has unacceptable quality [13]) and the identification of components that fall outside quality limits. We denote M'_c as the critical value of M' . Validated metrics are used to identify components with unacceptable quality. The purpose of control is to allow software managers to identify software that has unacceptable quality sufficiently early in the development process to take corrective action. For example, $M'_c = 3$ would be used as a critical value of cyclomatic complexity to discriminate between components that contain errors and those that do not.

Control also involves the tracking of the quality of a component over its life cycle. For example, if M' is cyclomatic complexity, an increase from 3 to 10, as the result of a design change, would be used to indicate possible degradation in quality. M' is shown as collected at point 4 in Figure 1 and used for control at point 5.

Quality Prediction

Quality prediction is a forecast of the value of F at time T_2 based on the values of M'_1, M'_2, \dots, M'_n for components 1, 2, \dots, n at time

T1, where "time" could be computer execution time, labor time or calendar time. Validated metrics (e.g., size, complexity) are used during the design phase to make predictions of test or operational phase factors (e.g., error count). The purpose of prediction is to provide software managers with a forecast of the quality of the operational software and to flag components for detailed inspection whose predicted factor values are greater than (or less than) the target values (determined from requirements analysis). M' is shown as collected at point 4 in Figure 1 and used for prediction at point 5.

Validity Criteria

Validity criteria provide the rationale for validating metrics; they are the specific quantitative relationships that are hypothesized to exist between factors and metrics. Validity criteria, in turn, are based on the principle of validity, which defines the general quantitative relationship between factors and metrics that must exist for the validity criteria to be applied. First we provide definitions relating to the principle of validity. Then we define the principle of validity. Last we define each validity criterion and provide an example of its application.

Definitions:

R[M]: Relation R on vector M for V[P1, T1, M] (1)

R[F]: Relation R on vector F for V[P1, T2, F] (2)

R[M']: Relation R on vector M' for A[P2, T1, M'] (3)

R[F']: Relation R on vector F' for A[P2, T2, F'] (4)

where R could be, for example, an order relation like:

Magnitude[M₁<M₂...<M_n] and Magnitude[F₁<F₂...<F_n] involving n values (data points) for M and F.

Principle of Validity:

IF R[M] \Longleftrightarrow R[F]

is validated statistically with confidence level α and, for certain validity criteria, with threshold value β_1 ,

THEN {R[M] \Longleftrightarrow R[F]} \implies {R[M'] \implies R[F']}? (5)

In other words, does the mapping M \Longleftrightarrow F, validated on Project 1, imply a mapping M' \implies F' on Project 2? We assume (5) to be true at point 5 in Figure 1. Once F' is collected at point 6, we revalidate (or invalidate) (5) by repeating the validation test using aggregated M and M' validated with respect to aggregated F and F' at point 7.

We note that a metric may be valid with respect to certain validity criteria and invalid with respect to other criteria. Each validity

criterion supports one or more of the quality functions assessment, control and prediction, which were described above. The validity criteria -- association, consistency, discriminative power, tracking, predictability and repeatability -- are applied at point 3 of Figure 1. The particular criteria that are used depend on the quality functions (one or more) that are to be supported.

The validation procedure requires that threshold values β_1 be selected for certain validity criteria. The criterion used for selecting these values is reasonableness (i.e., judgement must be exercised in selecting values to strike a balance between the one extreme of causing an M, which has a high degree of association with F, to fail validation and the other extreme of allowing an M of questionable validity to pass validation).

A short simple numerical example follows the definition of each validity criterion for the purpose of illustrating the basic concepts of the validity criteria. For illustrative purposes, F is error count and M is cyclomatic complexity, or complexity for short, in the examples. Also, to keep the examples simple, we use small sample sizes; these sample sizes would not be acceptable in practice. As noted previously, given {F} and {M}, it is possible to have an M_j in {M} predict multiple F_s in {F} or to have an F_1 in {F} predicted by multiple M_s in {M}. However, in order to simplify the examples, only the one-to-one case will be illustrated.

Association:

The variation in F explained by the variation in M, which is given by R^2 (coefficient of determination), where R is the linear correlation coefficient, must exceed a specified threshold, or

$$R^2 > \beta_a, \text{ with specified } \alpha. \quad (6)$$

This criterion assesses whether there is a sufficient linear association between F and M to warrant using M as an indirect measure of F. This criterion supports the quality assessment function as follows:

If the elements of vector M, corresponding to components 1,2, ...,n, are ordered by magnitude, as illustrated in Table 1, can we infer a linear ordering of F with respect to M for the purpose of assessing differences in component quality? In other words does the following hold?

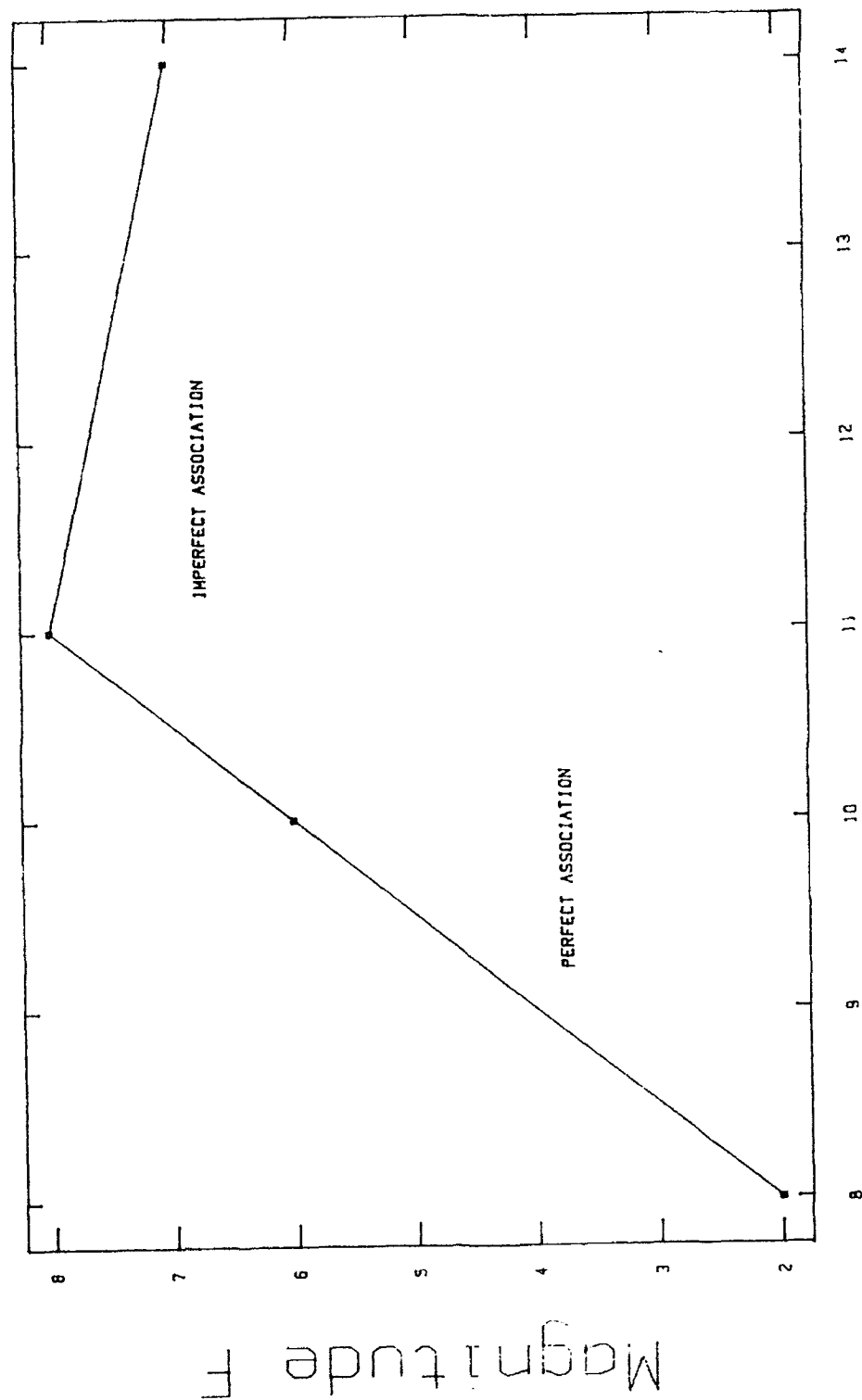
$$\begin{aligned} \text{magnitude}[M_1 < M_2 \dots < M_1 \dots < M_n] &\Longleftrightarrow \\ \text{magnitude}[F_1 < F_2 \dots < F_1 \dots < F_n] & \end{aligned} \quad (7)$$

F_i

$$\text{and } (M_{i+1} - M_i) \propto (F_{i+1} - F_i) \text{ for } i = 1, 2, \dots, n-1.$$

The data of Table 1 are plotted in Figure 2 to contrast perfect with imperfect association.

Association Validity Criterion



Magnitude M

COMPONENT

1

2

3

4

FIGURE 2

Table 1
(Validation Project)

Component	M (Magnitude)	M (Rank)	F (Magnitude)	F (Rank)
1	8	1	2	1
2	10	2	6	2
3	11	3	8	4
4	14	4	7	3

Since there is seldom perfect linear magnitude ordering between F and M (i.e., $R = 1.0$), we use (6) to measure the degree to which (7) holds. For example, if $R = .9$ and $\alpha = .05$, then 81% of the variation in F (error count) is explained by the variation in M (complexity), with an acceptable confidence level. If this relationship is demonstrated over a representative sample of components, and if β_a has been established as .7, we could conclude that M is associated with F and can be used to compare magnitudes of complexity obtained from different components to assess the degree to which they differ in quality (e.g., the difference in complexity magnitude between component 2 and component 1 (10 - 8) is proportional to their differences in quality in Table 1).

The resultant M' would be used to assess differences in the quality of components on the application project.

Consistency:

The rank correlation coefficient r between F and M must exceed a specified threshold, or

$$r > \beta_c, \text{ with specified } \alpha. \quad (8)$$

This criterion assesses whether there is sufficient consistency between the ranks of F and the ranks of M to warrant using M as an indirect measure of F [9]. This criterion supports the quality assessment function as follows:

If the elements of vector M, corresponding to components 1, 2, ..., n, are ordered by rank, as illustrated in Table 1, can we infer an ordering of F with respect to M for the purpose of assessing the rank order of component quality? In other words does the following hold?

$$\begin{aligned} \text{Rank}[M_1 < M_2 \dots < M_n] &\Longleftrightarrow \\ \text{Rank}[F_1 < F_2 \dots < F_n] \end{aligned} \quad (9)$$

The data of Table 1 are plotted in Figure 3 to contrast perfect with imperfect consistency for the same set of components.

Since there is seldom perfect rank ordering between F and M (i.e., $r = 1.0$), we use (8) to measure the degree to which (9) holds. For example, if $r = .8$ and $\alpha = .05$, there is an 80% ranking between F and M, with an acceptable confidence level. If this relationship is

Consistency Validity Criterion

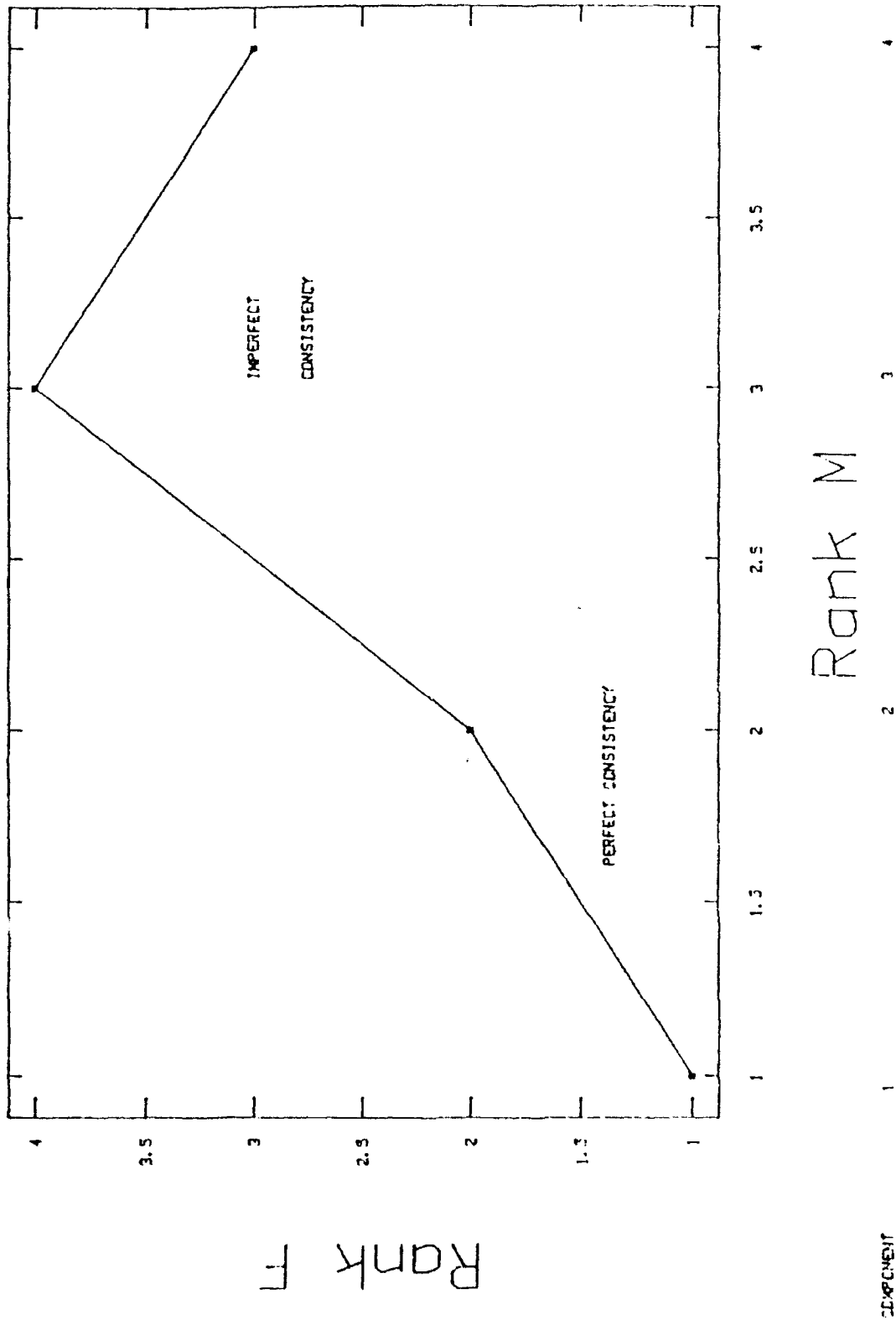


FIGURE 3

demonstrated over a representative sample of components, and if F_c has been established as .7, we could conclude that M is consistent with F and can be used to compare ranks of complexity obtained from different components to assess the degree to which they differ in relative quality (e.g., component 2 quality is lower (higher complexity) than component 1 quality in Table 1).

The resultant M' would be used to assess relative quality of components on the application project.

Discriminative Power:

The critical value of a metric M_c must be able to discriminate, for a specified F_c , between elements (components 1,2,...,i,...,n) of vector F [17], in the following way:

$$\begin{aligned} M_i > M_c &\iff F_i > F_c \text{ and} \\ M_i \leq M_c &\iff F_i \leq F_c \end{aligned} \quad (10)$$

for $i = 1,2,\dots,n$, with specified α .

This criterion assesses whether M_c has sufficient discriminative power to warrant using it as an indirect measure of F_c . This criterion supports the quality control function as follows:

Would M_c , as illustrated in Table 2, partition F , for a specified F_c , as defined in (10)? For example, the data from Table 1 is used in Table 2, with $M_c = 10$ and $F_c = 2$. We see that discriminative power is not perfect in Table 2 (i.e., $O_{11} \neq 0$). If it is desired to flag components with more than two errors ($F > F_c$) for detailed inspection, and if $M'_c = 10$ (complexity) is validated, it would be used on the application project to control quality (i.e., discriminate between acceptable and unacceptable components), as shown in Figure 4. One purpose of Figure 4 is to identify trends in quality (e.g., a persistent case of components being in the unacceptable zone).

Table 2
(Validation Project)

$M_c = 10$ $F_c = 2$	$M \leq M_c$	$M > M_c$
$F \leq F_c$	$O_{11} = 1$	$O_{12} = 0$
$F > F_c$	$O_{21} = 1$	$O_{22} = 2$

O_{ij} = count of observations in cell i,j .

O_{11}, O_{22} : correct classifications.

O_{12}, O_{21} : incorrect classifications.

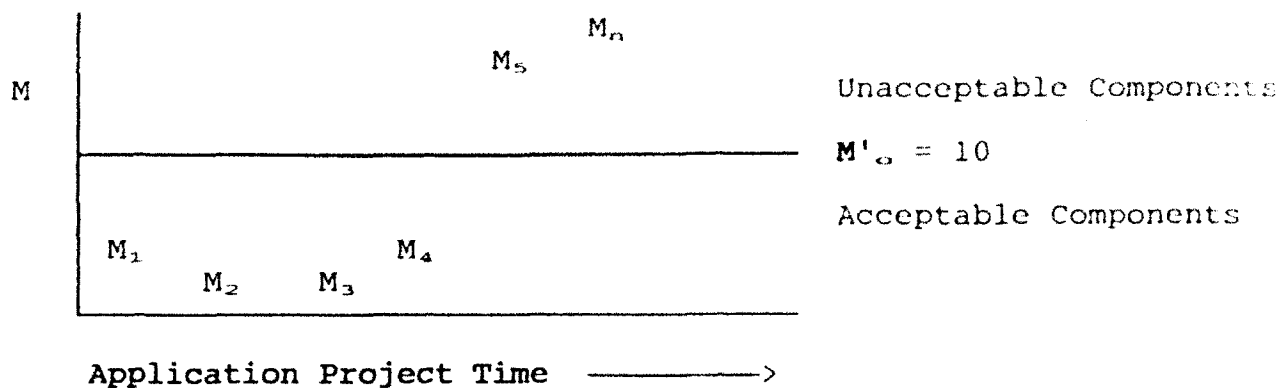


Figure 4. Application of Metrics to Quality Control (discriminative power) for Components 1,2,...,n

Since there is seldom a perfect discriminator M_α for F_α (i.e., $O_{11} = O_{21} = 0$ in Table 2), we use an appropriate statistical method (e.g., chi-square contingency table [7,8,12]) and representative sample of components to measure the degree to which (10) holds.

Tracking:

M must change in unison with F , for a given component i , at times $T_1, T_2, \dots, T_j, \dots, T_m$ as follows:

$$\begin{aligned} M_1(T_{j+1}) > M_1(T_j) &\iff F_1(T_{j+1}) > F_1(T_j) \text{ and} \\ M_1(T_{j+1}) = M_1(T_j) &\iff F_1(T_{j+1}) = F_1(T_j) \text{ and} \\ M_1(T_{j+1}) < M_1(T_j) &\iff F_1(T_{j+1}) < F_1(T_j) \end{aligned} \quad (11)$$

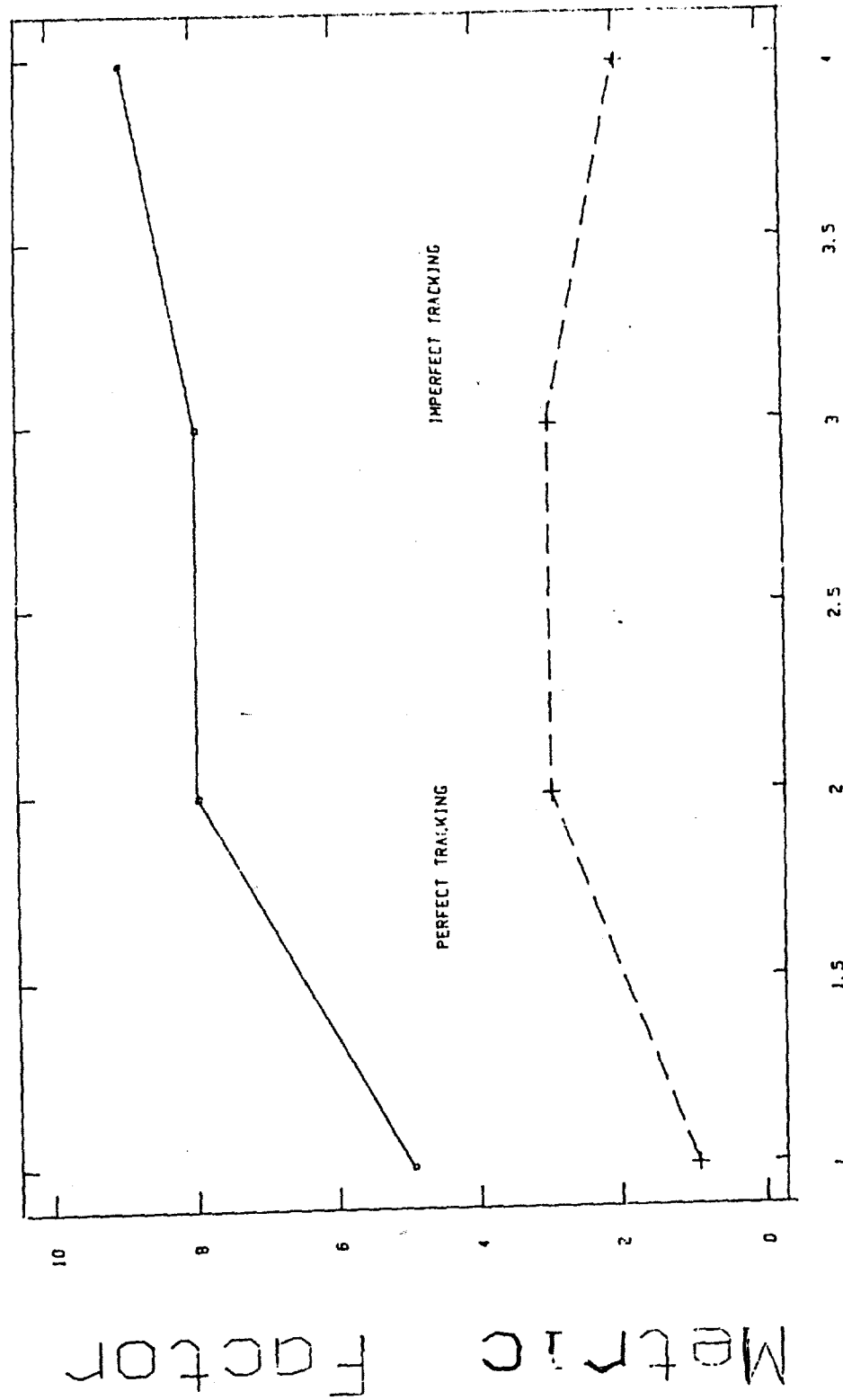
with specified α .

This criterion is illustrated graphically in Figure 5 to contrast perfect with imperfect tracking, where factor and metric values are plotted against project time.

This criterion assesses whether M is capable of tracking changes in F (e.g., as a result of design changes) to a sufficient degree to warrant using M as an indirect measure of F . This criterion supports the quality control function as follows:

Would changes in M track changes in F as defined in (11)? If M is validated, then a vector $M'_1(T_j)$ consisting of the values $M'_1(T_1), M'_1(T_2), \dots, M'_1(T_j), \dots, M'_1(T_m)$ of component i , measured at times $T_1, T_2, \dots, T_j, \dots, T_m$ would be used to track quality on the application project. For example, if complexity M'_1 is valid for tracking error count F , M'_1 would be used as shown in Figure 6, where quality increases from T_1 to T_2 , stays the same from T_2 to T_3 , and decreases thereafter.

Tracking Validity Criterion (Component 1)



Validation Project Time

FIGURE 5

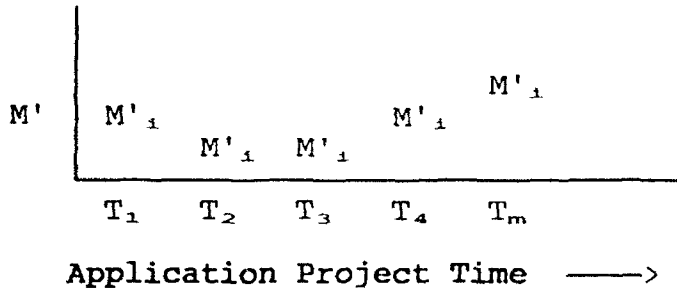


Figure 6. Application of Metrics to Quality Control (tracking) for Component i at Times 1,2,...,m

Since there is seldom perfect tracking of F by M, we use an appropriate statistical method (e.g., binary sequences test [8]) and representative sample for component i to measure the degree to which (11) holds.

Predictability:

A function of M, $f(M)$, where M is measured at time T1, must predict F, measured at time T2, with an accuracy β_p , or

$$\left| \frac{Fa_{T2} - Fp_{T2}}{Fa_{T2}} \right| < \beta_p \quad (12)$$

where Fa_{T2} is the actual value and Fp_{T2} is the predicted value.

This criterion is illustrated graphically in Figure 7 to contrast perfect with imperfect prediction, where $f(M)$, formulated at T1, will either turn out to be equal to Fa at T2 (perfect Predictability), or be equal to $Fp+$ or $Fp-$ (imperfect Predictability).

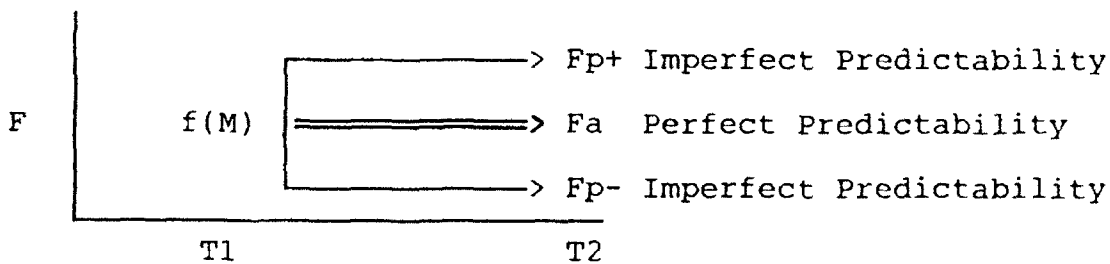


Figure 7. Application of Metrics to Quality Prediction (Predictability) for a Component

This criterion assesses whether $f(M)$ can predict F with required accuracy. This criterion supports the quality prediction function as follows:

If (12) holds, would the following hold?

$$Fp_{T2} = f(M_{T1}) \implies Fp'_{T2} = f(M'_{T1}) \quad (13)$$

where vector $Fp_{T2} = [F_1, F_2, \dots, F_n]_{T2}$ and vector $M_{T1} = [M_1, M_2, \dots, M_n]_{T1}$ for components $1, 2, \dots, n$, and Fp'_{T2} and M'_{T1} are similarly defined. In other words do we have:

$$\left| \frac{Fa'_{T2} - Fp'_{T2}}{Fa'_{T2}} \right| < \beta_p. \quad (14)$$

For example, if a function f , relating error count with complexity can be identified (e.g., regression analysis) that is a good predictor of F (i.e., satisfies (12)), then we would use the same f as the predictor of F' to predict error count from complexity on the application project.

Since there is seldom a perfect f , (i.e., $Fp_{T2} = Fa_{T2}$), we use (12) to measure the degree to which f predicts F .

Repeatability:

The success rate of validating M for a given validity criterion i must satisfy:

$$N_{i,s}/N_i > \beta_{i,s} \quad (15)$$

where $N_{i,s}$ is the number of validations of M for criterion i and N_i is the total number of trials for criterion i .

This criterion assesses whether M can be validated on a sufficient percentage of trials to have confidence that it would be a dependable indicator of quality in the long run. We use "trials" because validation could be performed with respect to projects, applications, components, or some other appropriate entity.

Metrics Validation Process

Given that there must be a validation project V and an application project A , as shown in Figure 1, this requirement gives rise to what we call the "fundamental problem in metrics validation". This problem arises because there could be significant time lags, product and process differences, and differences in goals and environments [5] between the following phases of the validation process (see Figure 1):

- 1) $V[P1, T1, M]$ and $V[P1, T2, F]$
- 2) $V[P1, T2, F]$ and $A[P2, T1, M']$
- 3) $A[P2, T1, M']$ and $A[P2, T2, F']$.

An important characteristic of the methodology is expressed by the following:

$$\begin{aligned} \text{IF } & V[P1, T1, M] \iff V[P1, T2, F] \\ \text{THEN } & A[P2, T1, M'] \implies A[P2, T2, F']. \end{aligned} \quad (16)$$

From (16) it follows that at point 3 in Figure 1, M is validated in V . Whether M' will actually be valid in A will not be known until point

7. Thus it is worthwhile to discuss some of the practical difficulties of adhering to (16) and possible remedies.

With respect to 1), the product or process may have changed so much between T1 and T2 that M collected at T1 may no longer be representative of F. If this is the case, M should be collected again at T2 to validate against F. The advantage of collecting M at T1 is that it may be easier and less expensive than at T2 because M can be collected as a by-product of compilation and design and code inspections.

The same considerations apply with respect to 3) except now the concern is with whether M' collected at T1 should be used for revalidation at T2. However, note that it is mandatory that M' be collected at T1 to have an early indication of possible quality problems (that is a key concept of our methodology!).

With respect to 2), we can achieve a degree of stability in the validation process if the following procedure is employed:

- a) Select V and A to be as similar as possible with respect to application and development environments.

With respect to 1), 2) and 3) considered jointly, we can achieve a degree of stability in the validation process if a) is employed plus the following two additional procedures:

- b) Select the same life cycle phase for T1 in V and A.
- c) Select the same life cycle phase for T2 in V and A.

We recognize that it may be infeasible to implement a), b) and c). If this is the case, it means there is a higher risk that M validated at point 3 in Figure 1 will not remain valid at point 5.

NON-PARAMETRIC STATISTICAL METHODS FOR METRICS VALIDATION

Non-parametric statistical methods are used to support metrics validation because these methods have important advantages over parametric methods. Indeed it would be infeasible to validate metrics in many situations without their use. This is the case because the assumptions that must be satisfied to employ non-parametric methods are less demanding than those that apply to parametric methods. This might lead to the conclusion that non-parametric methods are less rigorous than parametric methods. Despite this possible perception, non-parametric methods allow us to develop very useful order relations concerning the relative quality of components. The validity criteria which use non-parametric methods are shown in Table 3. The advantages of non-parametric methods over parametric methods, which are important for metrics validation, are the following:

- o Given the noisiness of metrics data, the fact that the assumptions are less restrictive is a big advantage.
- o No assumption is necessary about distribution (e.g., data does not have to be normally distributed).

- o We can use the nominal scale (i.e., component A is high quality, component B is low quality) and location statistics like the median [11]. The Discriminative Power validity criterion is based on this measurement property. Similarly, we can use the nominal scale to indicate whether an incremental change in a metric tracks (yes, no) an incremental change in a factor. The Tracking validity criterion is based on this measurement property.
- o We can use the ordinal scale (i.e., component A is higher quality than component B) and order statistics, like ranks. The Consistency validity criterion is based on this measurement property. For example, ranks of random variables [3] can be used rather than the values themselves, thus relaxing the assumptions about data relationships (e.g., linearity) while providing a measure of quality (e.g., ranking of components) that is useful to the software manager. In other words the fact that the data is not as "well-behaved" as we might believe it should be does not necessarily mean that it is less useful. In fact, when we consider that many useful applications of metrics can be derived from the ability to classify components as being "higher quality" or "lower quality", we realize that the information provided by non-parametric analysis is supportive of this approach.

Despite the advantages of non-parametric methods, certain validity criteria lend themselves to the use of parametric methods. These are shown in Table 3. Association, which measures the difference in component quality, uses the interval scale. Predictability uses the interval scale to predict a factor value and the ratio scale for measuring prediction accuracy. Lastly, Repeatability uses the ratio scale for measuring metric validation success.

Appendix A summarizes quality function, validity criterion, purpose of valid metric, and statistical method.

Table 3

Validity Criteria Properties

Criterion	Scale	Method	Measurement Property
Association	Interval	Parametric	Difference
Consistency	Ordinal	Non-parametric	Higher/Lower
Discriminative Power	Nominal	Non-parametric	High/Low
Tracking	Nominal	Non-Parametric	Increment
Predictability	Interval, Ratio	Parametric	% Accuracy
Repeatability	Ratio	Parametric	% Success

EXAMPLE OF VALIDATING METRICS

The following example is provided to illustrate the validation of M with F and the identification of an M_c which would be used in the quality control function. Also we show how to conduct a cost sensitivity analysis on M_c in order to identify its optimal value (i.e., the minimum cost M_c across a range of assumptions about the cost of using M_c).

The data used in the example validation tests were collected from actual software projects. The Discriminative Power validity test is illustrated.

Purpose of Metrics Validation

The purpose of this validation is to determine whether cyclomatic number (complexity (C)) and size (number of source statements (S)) metrics, either singly or in combination, could be used to control the factor reliability, as represented by the factor error count (E). A summary of the data is shown in Table 4 and the detailed data listing can be found in Appendix B.

Table 4

Project Application	Procedures (with errors)	Statements	Errors
1 String Processing	11 (5)	136	10
2 Directed Graph Analysis	31 (12)	430	27
3 Directed Graph Analysis	1 (1)	13	1
4 Data Base Management	69 (13)	1021	26
112 (31)		1600	64

Number of procedures: 112 total, 31 with errors, 81 with no errors.

Number of source statements: 2007 total, 1600 included in metrics analysis.

Language : Pascal on all projects.

Programmer: Single programmer. Same programmer on all projects.

Using the conventions of Figure 1, the following is the notation applicable to this example:

Metric: C , S collected at point 1, Figure 1.

Factor: E , collected at point 2, Figure 1.

Critical Value of Metric: C_c , S_c validated at point 3, Figure 1.

V [Projects 1,2,3,4; Design; C , S]

V [Projects 1,2,3,4: Test; E]

Discriminative Power Validity Test

We divide the data into four categories, as shown in Table 5, according to a critical value of C , C_c , so that a chi-square test can be performed to determine whether C_c can discriminate between procedures with errors and those with no errors [4].

Table 5

Contingency Table

	Complexity ≤ 3	Complexity > 3	
no Errors	75	6	81
Errors	10	21	31
	85	27	112

From the high value of chi-square (41.60) (see Table 6) and the very small significance level ($1.26E-10$) in the samples, we infer that $C_c = 3$ could discriminate between procedures with errors (low quality software) and those without errors (high quality software).

Table 5 shows how good a job $C_c = 3$ does to discriminate between procedures with errors and procedures with no errors: 75 of 81 with no errors and 21 of 31 with errors are correctly classified.

Table 6

Projects 1, 2, 3 and 4

12 Procedures (81 with no errors, 31 with errors)

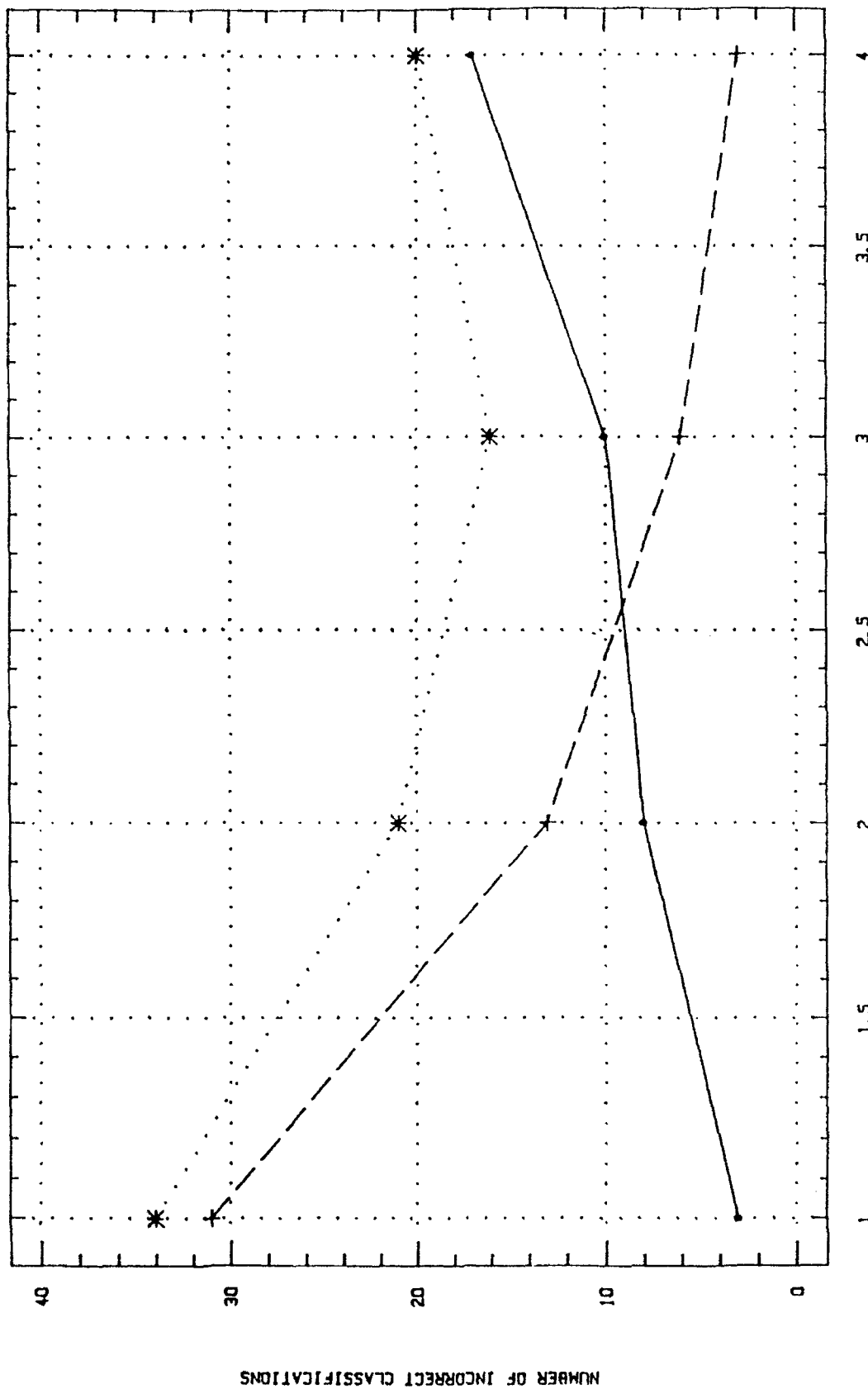
χ^2	α
22.32	$2.30E-6$
32.14	$1.44E-8$
41.60	$1.26E-10$
26.80	$2.26E-7$

Sensitivity Analysis of Critical Value of Complexity

In order to see how good a discriminator C_c is for this example, we observe the number of misclassifications that result for various values of C_c : 1) Type 1 ("error procedures" classified as "no error procedures") and 2) Type 2 ("no error procedures" classified as "error procedures"). This is shown in Figure 8. As C_c increases, Type 1 misclassifications increase because an increasing number of high complexity procedures, many of which have errors, are classified as having "no errors". Conversely, as C_c decreases, Type 2 misclassifications increase because an increasing number of low complexity procedures, many of which have no errors, are classified as having "errors". The total of the two curves represents the "misclassification function". It has a minimum at $C_c = 3$, which is the value given by the chi-square test (see Table 6). The chi-square test will not always produce the optimal C_c but the value should be close to optimal.

—•— TYPE 1
 - - - TYPE 2
 * * * TOTAL

INCORRECT CLASSIFICATION
 (COMPLEXITY)



CRITICAL VALUE OF COMPLEXITY

Figure 8

The foregoing analysis assumes that the costs of Type 1 and Type 2 misclassifications are equal. This is usually not the case since the consequences of not finding an error (i.e., concluding that there is no error when, in fact, there is an error) would be higher than the other case (i.e., concluding that there is an error when, in fact, there is no error). In order to account for this situation, the number of Type 1 misclassifications, for given values of C_e , is multiplied by $C1/C2$ ($C1/C2 = 1, 2, 3, 4, 5$), which is the ratio of the cost of Type 1 misclassification to the cost of Type 2 misclassification. These values are added to the number of Type 2 misclassification to produce the family of five "cost" curves shown in Figure 9. Naturally, with the higher cost of Type 1 misclassifications taking effect, the optimal C_e (i.e., minimum cost) decreases. However, even at $C1/C2 = 5$, $C_e = 3$ is a reasonable choice.

A Contingency Table was also developed for S, leading to $S_e = 13$. The same type of sensitivity analysis was performed on S_e . It was found that the optimal $S_e = 15$, as opposed to $S_e = 13$, as given by the chi-square analysis.

We conclude that C and S are valid with respect to the Discriminative Power criterion and either could be used to distinguish between acceptable ($C \leq 3$, $S \leq 13$) and unacceptable quality ($C > 3$, $S > 13$) for this and similar applications when this data can be collected. However, only one is needed (i.e., C is highly correlated with S). It should be noted that it is less expensive to collect S than C.

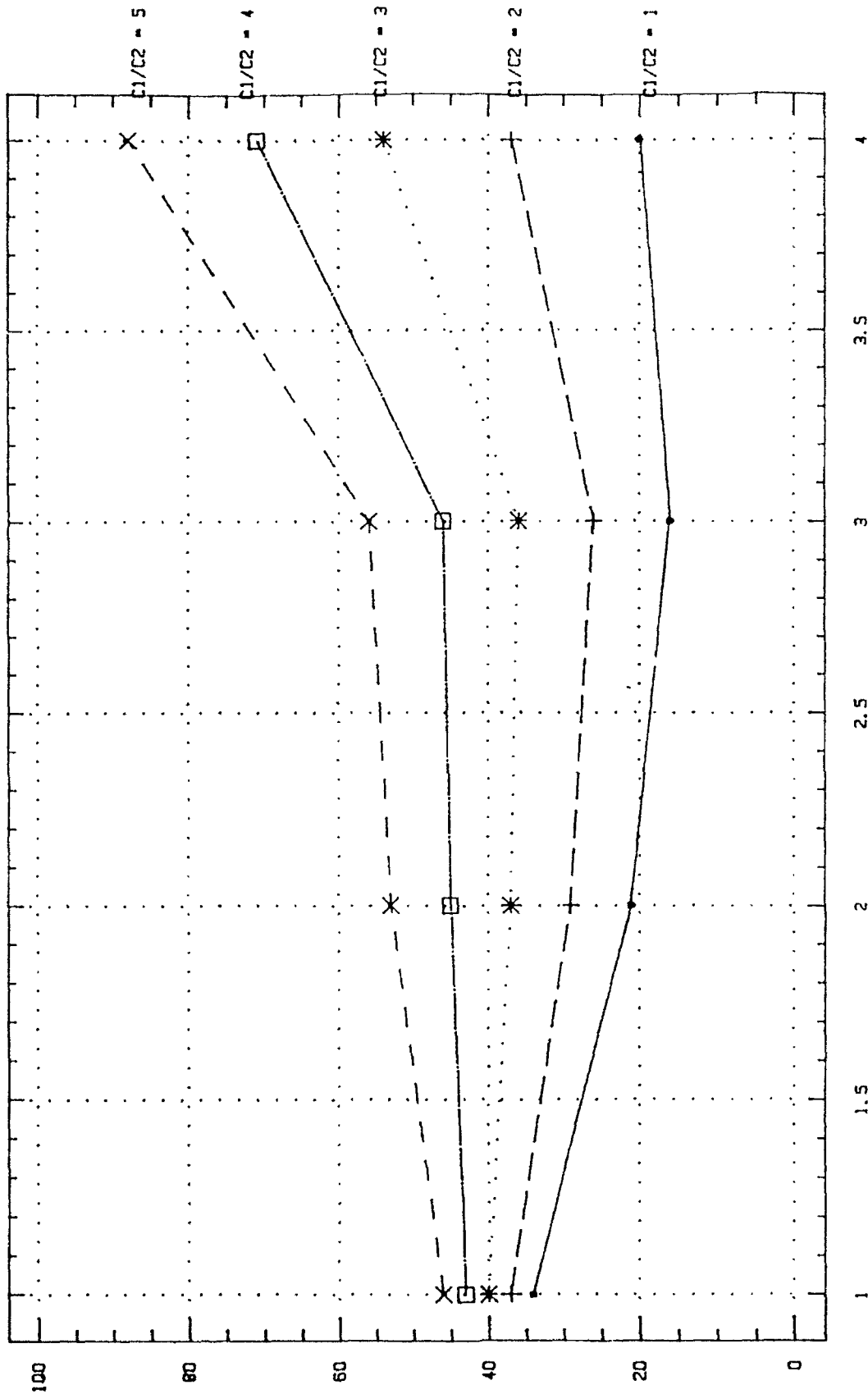
SUMMARY AND FUTURE RESEARCH

We described and illustrated a comprehensive metrics validation methodology that has six validity criteria, which support the quality functions of assessment, control and prediction. Six criteria were defined and illustrated: association, consistency, discriminative power, tracking, predictability and repeatability. These criteria are important because they provide a rationale for validating metrics; in practice, this rationale is frequently lacking in the selection and application of metrics. With validated metrics we have a basis for making decisions and taking actions to improve the quality of software. We showed that quality factors, metrics and functions can be integrated with our metrics validation process. We developed a framework which pulls together the concepts and definitions of quality factor, quality metric, validated metric, quality function, validity criteria, and the metrics validation process. We showed that non-parametric statistical methods play an important role in evaluating whether metrics satisfy the validity criteria. An example of the application of the methodology was presented for the discriminative power validation criterion. The discriminative power criterion allows the metrics user to control the production of highly reliable software by providing thresholds of acceptable quality.

Future research is needed to extend and improve the methodology by finding an answer to the following question:

o To what extent are metrics that have been validated on one project, using our criteria, valid measures of quality on future projects -- both similar and different projects?

COST OF INCORRECT CLASSIFICATION
(COMPLEXITY)



CRITICAL VALUE OF COMPLEXITY

Figure 9

acknowledgements

We thank the referees for their many useful comments and suggestions that we believe have greatly improved the paper. We acknowledge the support provided for this research project by the Naval Surface Warfare Center and the Army Operational Test and Evaluation Center. We also thank the members of the IEEE Standard for a Software Quality Metrics Methodology Working Group for many useful discussions and debates that helped inspire this work.

REFERENCES

-] A. J. Albrecht and J. E. Gaffney, Jr., "Software Function, Source Lines of Code, and Development Error Prediction: A Software Science Validation", IEEE Transactions on Software Engineering, Vol. SE-9, No. 6, November 1983, pp. 639-648.
-] Albert L. Baker, et al, "A Philosophy for Software Measurement", The Journal of Systems and Software, Vol. 12, No. 3, July 1990, pp. 277-281.
-] Victor R. Basili, Richard. W. Selby, Jr., and Tsai-Yun Phillips, "Metric Analysis and Data Validation Across Fortran Projects", IEEE Transactions on Software Engineering, Vol. SE-9, No. 6, November 1983, pp. 652-663.
-] Victor R. Basili, and David H. Hutchens, "An Empirical Study of a Syntactic Complexity Family", IEEE Transactions on Software Engineering, Vol. SE-9, No. 6, November 1983, pp. 664-672.
-] V. R. Basili and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments", IEEE Transactions on Software Engineering, Vol. 14, No. 6, June 1988, pp. 759-773.
-] Martin E. Bush and Norman E. Fenton, "Software Measurement: A Conceptual Framework", The Journal of Systems and Software, Vol. 12, No. 3, July 1990, pp. 223-231.
-] David N. Card, Gerald T. Page, and Frank E. McGarry, "Criteria for Software Modularization", Proceedings of the 8th International Conference on Software Engineering, August 28-30, 1985, pp. 372-377.
-] W. J. Conover, Practical Nonparametric Statistics, John Wiley & Sons, Inc., 1971.
-] S. D. Conte, H. E. Dunsmore and V. Y. Shen, Software Engineering Metrics and Models, The Benjamin/Cummings Publishing Company, Inc., 1986.
- 0] Leonardo Felician and Graziella Zalateu, "Validating Halstead's Theory for Pascal Programs", IEEE Transactions on Software Engineering, Vol. 15, No. 12, December 1989, pp. 1630-1632.

- [11] Norman E. Fenton and Austin Melton, "Deriving Structurally Based Software Metrics", The Journal of Systems and Software, Vol. 12, No. 3, July 1990, pp. 177-187.
- [12] Jean Dickinson Gibbons, Nonparametric Statistical Inference, McGraw-Hill Book Company, 1971.
- [13] IEEE Standard for a Software Quality Metrics Methodology (Draft), P-1061/D21, April 1, 1990.
- [14] IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Std 729-1983.
- [15] IEEE Glossary of Software Engineering Terminology (Draft), P729/610.12/D8, March 30, 1990.
- [16] Adam A. Porter and Richard W. Selby, "Empirically Guided Software Development Using Metric-Based Classification Trees", IEEE Software, Vol. 7, No. 2, March 1990, pp. 46-54.
- [17] Elaine J. Weyuker, "Evaluating Software Complexity Measures", IEEE Transactions on Software Engineering, Vol. 14, No. 9, September 1988, pp. 1357-1365.

APPENDIX A

Quality Function	Validity Criterion	Purpose of Valid Metric	Statistical Method
Quality Assessment	Association	Assess differences in quality	<ol style="list-style-type: none"> 1. Coeff. of Determination $R^2 > \beta_{..}$. 2. H_0: Population Correlation Coeff. = 0. 3. H_0: Population Correlation Coefficient $> \sqrt{\beta_{..}}$. 4. Linear Partial Correlation Coeff. (Metric Normalization. Accounting for Size). 5. Population Correlation Coefficient Confidence Interval. 6. Factor Analysis (Tests of Independence).
Quality Assessment	Consistency	Assess relative quality	<ol style="list-style-type: none"> 1. Rank Correlation Coefficient $r > \beta_{..}$.
Quality Control	Discriminative Power	Control Quality (discriminate between high and low)	<ol style="list-style-type: none"> 1. Mann-Whitney Comparison of Average Ranks of Two Groups of components. 2. Chi-square Contingency Table for Finding Critical Value of Metric. 3. Short-Cut Technique for Finding Critical Value of Metric: Maximize $O_{11}O_{22}$. 4. Sensitivity Analysis of Critical Value of Metric. 5. Krusal-Wallis Test of Average Metric Rank Per Given Value of Quality Factor. 6. Discriminant Analysis (Use of a Single Metric's Mean as Discriminator).
Quality Control	Tracking	Control quality (track changes)	<ol style="list-style-type: none"> 1. Binary Sequences Test and Wald-Wolfowitz Runs Test.

APPENDIX A (Continued)

Quality Function	Validity Criterion	Purpose of Valid Metric	Statistical Method
Quality Prediction	Predictability	Predict quality	<ol style="list-style-type: none"> 1. Scatter Plot to Investigate Linearity. 2. Linear Regression. <ol style="list-style-type: none"> a. Test Assumptions. b. Examine Residuals. 3. Find Confidence and Prediction Intervals. 4. Test for Predictability $< \text{Threshold } (\beta_p)$ and Repeatability $> \text{Threshold } (\beta_{rs})$. 5. Non-linear Regression. 6. Multiple Linear Regression. <ol style="list-style-type: none"> a. Test Assumptions. b. Examine Residuals. c. Test for Predictability $< \text{Threshold } (\beta_p)$ and Repeatability $> \text{Threshold } (\beta_{rs})$.
All Quality Functions	Repeatability	Ensure metric validated with specified success rate	Ratio of Validations to Total Trials $> \text{Threshold } (\beta_{rs})$.

APPENDIX B

C: Complexity, S: Number of Source Statements (excluding comments)
E: Error Count

Procedures with No Errors

C	S	E	Project	C	S	E	Project
2	6	0	1	1	3	0	4
1	8	0	1	1	3	0	4
1	11	0	1	1	3	0	4
1	4	0	1	1	5	0	4
2	18	0	1	1	5	0	4
3	15	0	1	1	6	0	4
1	3	0	2	1	9	0	4
1	3	0	2	1	6	0	4
1	3	0	2	1	8	0	4
1	3	0	2	1	9	0	4
1	3	0	2	1	9	0	4
1	3	0	2	2	4	0	4
1	3	0	2	2	7	0	4
1	3	0	2	2	9	0	4
1	5	0	2	4	56	0	4
1	5	0	2	1	24	0	4
1	5	0	2	2	13	0	4
1	13	0	2	2	13	0	4
1	3	0	2	2	10	0	4
1	3	0	2	2	9	0	4
1	3	0	2	2	12	0	4
1	3	0	2	5	21	0	4
1	3	0	2	5	49	0	4
1	3	0	2	3	19	0	4
1	3	0	2	4	20	0	4
1	2	0	4	2	6	0	4
1	2	0	4	2	12	0	4
1	7	0	4	2	9	0	4
1	5	0	4	2	10	0	4
1	7	0	4	1	21	0	4
1	5	0	4	4	21	0	4
1	5	0	4	3	11	0	4
1	5	0	4	2	13	0	4
1	5	0	4	3	14	0	4
1	4	0	4	7	19	0	4
1	3	0	4	2	15	0	4
1	3	0	4	2	10	0	4
1	3	0	4	2	17	0	4
1	3	0	4	3	19	0	4
1	3	0	4	3	15	0	4
1	3	0	4	2	15	0	4

APPENDIX B (Continued)

Procedures with Errors

C	S	E	Project	C	S	E	Project
2	14	1	1	4	26	1	2
6	26	5	1	16	94	8	2
5	7	2	1	2	13	1	3
5	21	1	1	6	83	1	4
2	6	1	1	5	28	1	4
1	3	1	2	8	37	5	4
1	11	1	2	3	13	2	4
1	8	1	2	3	16	1	4
2	15	3	2	7	34	1	4
8	45	3	2	5	24	1	4
4	18	1	2	4	18	3	4
6	54	3	2	5	35	2	4
2	34	2	2	13	49	5	4
4	19	1	2	4	19	1	4
5	30	2	2	4	27	1	4
				4	17	2	4

The Consolidated Experience Factory: An Approach for Instrumenting Systems Engineering

Robert L. Vienneau

Data & Analysis Center for Software
Kaman Sciences Corporation
258 Genesee Street
Utica, NY 13502

Presented at:

The 1992 Complex Systems Engineering Synthesis and Assessment
Technology Workshop

Abstract

This paper presents an organizational structure, the Consolidated Experience Factory (CEF), for instrumenting the system development process. Goals of interest to systems engineers and that can be satisfied by process and products metrics are briefly summarized. The goal-questions-metrics methodology has been developed in the context of software metrics for deriving individual metrics from high-level goals. Experience factories are organizations parallel to system development organizations that serve to define metrics, collect and validate data, analyze the data, and package the results in a usable form. The CEF is an organization for integrating the results of many experience factories.

Keywords

Systems engineering, Systems metrics, process metrics, software metrics, experience factories.

1. Introduction

This paper presents an approach for instrumentation, data collection, analysis, and improvement of the systems engineering process. This approach, known as the Consolidated Experience Factory (CEF), has been developed by Victor Basili of the University of Maryland in conjunction with the Data & Analysis Center for Software (DACS). The CEF was defined for software development and maintenance, but, as this paper shows, the approach is general enough to apply to systems development as a whole. Given the recognized importance of software in defense systems acquisition, the CEF attacks a crucial component of the problem addressed by this conference.

This paper is organized into six sections. Section 2 summarizes the systems engineering needs addressed by experience factories. Section 3 presents a method used in software, the Goals/Questions/Metrics paradigm, for deriving metrics from high level goals. Section 4 presents the concept of an experience factory, a logical or physical organization for measuring systems development, while Section 5 presents the Consolidated Experience Factory, an organization for integrating several experience factories. Finally, the concluding section discusses future work needed to implement this approach for instrumenting systems engineering.

2. Goal-Driven System Metrics

The first premise of the CEF approach is that measurements should be introduced into the development process to address specific goals. This premise may appear obvious, but some measurement projects for software have defined a comprehensive set of metrics such that any high-level goals were obscured. The Software Technology for Adaptable Reliable Systems (STARS) Data Collection Forms [IITRI 85] are a typical example. Furthermore, since a complete feasible set of metrics could not be known at the initiation of the field of software metrics, a failure to recognize this premise was probably a necessary stage in the field's development. The introduction of system metrics should take advantage of the expertise built up in software metrics over the last two decades and begin with a topdown approach.

The goals driving the system metrics collected on a particular project should be derived from the requirements of that project, the development organization, and the Navy sponsoring agencies. The particular metrics discussed in this paper are not those needed to test system requirements, such as throughput or functionality, but mainly process metrics to assist developers in project management. For example, a database can be developed characterizing the system development process in terms of the distribution of failures throughout the life cycle. A project manager can then compare how his data fits a typical project "fingerprint" at any point of time. Significant deviations will then suggest issues the

manager will want to address. A metric-based process can be used to insure systems are actually ready for scheduled reviews. The Army's Software Test and Evaluation Panel metrics are being used in this way [DOA-PAM-73XX]. A database of metric data on past projects conducted by the developing organization is required to fully implement a metric-based approach to systems development.

A system development organization will have goals that can be supported by metrics, in addition to the requirements of individual projects. The organization will need to develop the database of metric data that supports individual project managers. The organization will want to characterize their development process so as to support process improvements in a controlled manner. They will want to measure the effects of proposed techniques by controlled experiments. Once controlled experiments lead to a determination that a new technology should be adopted, measurement needs to be conducted to ensure that it is properly transferred to individual projects and that the expected benefits are obtained. The development organization needs to understand relationships between the system development process and the resulting products. All of these goals are most fully supported by more than raw metric data. The methods used in packaging metric-based models of the development process need to be carefully considered.

The Navy may want to consider the research needs of system engineering as a whole in their systems engineering program. If so, they should consider how to integrate the different research activities conducted on system engineering and avoid bottom-up isolated research activities. Experimentation and measurement should be used to evaluate and analyze systems research. The results of research should be refined and tailored for application environments and packaged such that they can be easily transferred to practice. The relationships between models of the development process and the products should be made clear. These high-level goals, and others, are addressed by the organization described in this paper.

3. The Goals-Question-Metrics Paradigm

Various top down approaches have been defined for deriving metrics from goals in software engineering. For example, Rome Laboratory has sponsored research that has developed a hierarchical framework for measuring specific quality factors [Bowen 85]. More suitable for a systems approach because of its generality is the Goals-Questions-Metrics (GQM) paradigm [Basili 90]. In fact, it was developed specifically to fit into the experience factory approach described in this paper.

The GQM paradigm is a mechanism for defining and interpreting measurable software (or system) goals. Goals are typically stated in the following format:

Purpose:

Analyze some **object** (e.g. process, product, experience model) for the purpose of **why** (e.g. characterization, evaluation, prediction, motivation, improvement)

Perspective

with respect to **focus** (e.g. cost, correctness, defect removal, reliability, user friendliness) from the point of view of **who** (e.g. user, customer, manager, developer, corporation)

Environment

in the following **context** (e.g. problem factors, people factors, resource factors, process factors).

The GQM paradigm then provides a structured process for generating measurement related questions from these goals. Each question, in turn, generates a set of metrics.

The GQM paradigm is not yet cookbook; it's application requires a knowledgeable person. As with all topdown approaches, its employment requires insight into reasonable lower level results. Furthermore, the templates developed so far are for software, not systems. Nevertheless, this paradigm is a very promising approach for system metrics.

4. The Experience Factory - An Organization for System Measurment

Experience factories provide an organization to apply measurement to system development to best meet the needs of individual projects and systems engineering organizations [Basili 89]. An experience factory is a logical organization supporting systems development by analyzing and synthesizing measured experiences, creating a repository of useful information, and supplying packaged results to various projects as they need them.

Input to the experience factory includes goals and data. The factory analyzes that data to characterize the environment and systems engineering methods, evaluate, predict, and improve. The outputs are packaged models.

Packaged results are central to an experience factory. It does not merely act as a repository of metric data. Packaging can best be displayed by means of examples. First, simple models might include simple formulas for prediction. For example, in software an equation might be given predicting total effort as a function of source lines of code, faults per thousand lines of code as a function of what methods are used, or total schedule length as a function of source lines. Packaged models of this sort are obviously useful for forecasting and planning purposes. More complicated packages might include

distributions of key variables, such as types of failures, at particular points in the life cycle. This second type of packaged result is useful for controlling a project during its development, as well as refining predictions. A third type of packaged result would be even more detailed. Models might be created, using a formal notation for the development process, showing the impacts of various combinations of methods on the distributions of key variables. With this sort of packaged result, the systems engineer can design his life cycle to meet his particular needs. The organization can also use these results in a scientific manner to assess the impacts of introducing proposed systems engineering techniques.

5. The Consolidated Experience Factory to Integrate Experience Factories

Many systems engineering groups might set up parallel experience factories. These factories will then be churning away producing packaged models that serve the needs of their groups more or less successfully. The discipline of systems engineering as a whole will be best served by integrating the results of the various experience factories. The Consolidated Experience Factory (CEF) will serve this purpose.

The CEF is an organization separate from any developer or experience factory. It receives input from the various experience factories and produces results of use to them. The CEF would answer questions like the following:

- What questions and metrics have organizations found useful for addressing specific goals?
- If a new method is introduced into a systems development organization, how might that effect packaged prediction models, based on the packaged results of other experience factories?
- What is the domain of applicability of various models? For example, what characteristics of a development organization determine which of many reliability models work the most successfully?
- Can "metamodels" be produced that combine models across experience factories? Will these metamodels be more useful than models tailored for an individual systems organization, especially as those organizations change?

6. Facing Many Questions

This paper has proposed a set of organizations (experience factories and the Consolidated Experience Factory) for instrumenting systems engineering to support the discipline's improvement in a measured, structured, and scientific manner. How can this vision be brought to pass?

First, this approach has only been defined for software in the past. The concept is general enough to apply to systems, but the details need to be redefined for systems. For example, templates have been defined in the past for applying the Goals/Questions/Metrics paradigm to software. New templates need to be created for systems.

Second, this vision can be fulfilled incrementally. The CEF can initially have a role of helping a small number of organizations create their own local experience factories. Perhaps, in keeping with this incremental strategy, these initial efforts need only focus on one aspect of the systems problem, software being the natural candidate.

Finally, details of sharing data must be defined. Questions of data confidentiality are not so important for a local experience factory and its development organization. But they are crucial for the interface between the Consolidated Experience Factory and the local experience factories. The concept of "packaged" results presents a new approach to successfully addressing this problem.

7. References

[Basili 89] Victor R. Basili, "The Experience Factory: Packaging Software Experience," Proceedings of the Fourteenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, November 1989.

[Basili 90] Victor R. Basili, "The Goal/Question/Metric Paradigm," white paper, University of Maryland, 1990.

[Bowen 85] T. P. Bowen, G. B. Wigle, and J. T. Tsai, Specification of Software Quality Attributes, RADC-TR-85-37 (3 volumes), Rome Air Development Center, February 1985.

[Chillarege 91] Ram Chillarege, "Defect Modelling," Fourteenth Minnowbrook Workshop on Software Engineering, Syracuse University, 1991.

[DOA-PAM-73XX] Test and Evaluation Guidelines, Volume 6, Software Test and Evaluation Guidelines (Draft), Department of the Army Pamphlet 73-XX, Headquarters, Department of the Army, September 1991.

[IITRI 85] IIT Research Institute, Interim Software Data Collection Forms Package, Data & Analysis Center for Software, 7 June 1985.

DESIGN CAPTURE METHODS

A Framework for the Engineering/Reengineering of Complex Systems

Bruce I. Blum

Johns Hopkins University/Applied Physics Laboratory
Laurel, MD 20723-6099
bib@aplcomm.jhuapl.edu

This paper presents a framework for classifying projects engaged in the engineering or reengineering of complex Navy systems. The object of this framework is to establish comparability among dissimilar projects and to aid in the transition to newer, more effective paradigms. The paper examines the problems faced by the Navy in the evolution of existing systems and analyzes the software process in the context of this challenge. A four-dimensional framework is defined for classifying projects and their data.

Introduction

The development, maintenance, and evolution of mission-critical, real-time systems is a very complex task. These systems are comprised of many communicating independent subsystems that must work together in a variety of stressful environments, some of which will be untested prior to the initial encounter. The systems are governed by the laws of physics, which may impose severe time constraints on both decisions and actions, and they must support the decision making of those who use and rely on them. These systems are composed of hardware, software, and humans, and each subsystem receives inputs from and directs outputs to other hardware, software, and human interfaces.

As demonstrated in Desert Storm, the Navy has developed excellent systems that perform very well. Many of these systems require more than a decade for development, and budgetary constraints are certain to limit the number of completely new systems that will be built in the near future. Thus, the challenge is to support the refinement of the existing systems by accepting modified or enlarged missions and exploiting emerging technologies while, at the same time, taking advantage of the Navy's significant investment in existing systems.

No formal models exist for a complete system. There are scientists and engineers who are specialists in selected areas such as sonar, radar, command and control, and weapons control systems. Yet, as the systems become further integrated and complex, we

This work was supported in part by the U. S. Navy, Space and Naval Warfare Systems Command (SPAWAR) under contract N00039-91-C-0001, task VMAR9 with the Office of Naval Research (ONR) and the Naval Surface Warfare Center (NSWC).

find problems that can be resolved only in a multidisciplinary setting. The consequences of interaction are difficult to anticipate, and there are few formal mechanisms for modeling the nonfunctional requirements associated with timing and resource utilization. In summary, we are confronted by "wicked problems" that can be neither resolved within a single discipline nor comprehended by a single individual. Our successes to date illustrate that viable solutions are within the state of the art. Our goal, therefore, is to continue this progress in a mode that emphasizes upgrade rather than retirement, reuse rather than replacement.

Although the task at hand may seem very restrictive, there are at least two reasons for optimism. First, one learns by experience, and the existence of complex, effective tactical systems is evidence that significant stores of knowledge exist. One problem with this knowledge base is that it is poorly organized. In software, for example, the knowledge of what the system does is embedded in the code, which describes—not what is done—but how it is done. One consequence is that only small software modifications are attempted; lacking system-level understanding, program managers avoid risk by localizing change. Such an approach is quite reasonable in an environment in which the next generation system will replace the present generation, but it is incompatible with a philosophy of evolving from one generation system to the next.

Fortunately, the second reason for optimism addresses this problem. A revolution in the way we perceive software is underway, and this makes the more rational capture and reuse of system knowledge possible. This transition to a new view of software, however, is still in its early stages, and most research in software engineering builds on the existing paradigm. Thus, when viewed in the context of how projects now are run, the new approach may seem speculative and high risk. Yet, because it offers such a potential for process improvement (as measured in productivity, adaptability, and quality), it is an important area of research. Further, because many researchers have been working with this paradigm for more than a decade, there is also a potential for near-term application.

Since 1980 I have been working with a modified paradigm for systems development. Most of my experience has been with interactive information systems, and a decade of activity has been carefully evaluated and reported [Enlb89, Blum90]. During the past few years I have been working in the domain of complex systems, albeit at the conceptual level. One product of this research is a new understanding of how we develop systems and the role that software plays in a system's development and evolution. This understanding now is sufficiently mature to permit the formulation of a framework for classifying activities that both aids in project comprehension and leads to the adoption of more effective methods.

This paper describes a general framework for engineering and reengineering that should aid in the evolution from one generation of system to the next. Because new systems will evolve from one generation to the next, the distinction between engineering and reengineering is fuzzy. For example, the introduction of a newly engineered component may have no immediate effect on the system's functionality, and the need to comply with existing interfaces may constrain a new development. Thus, rather than distinguish between these complementary activities, I shall treat them as one. Therefore, the objective of an engineering/reengineering framework is to guide the development and evolution of a system throughout its productive lifetime. It must consider the immediate concerns of the

managerial and technical staffs responsible for the engineering/reengineering activities, and it also must examine how their efforts can build a foundation for improved technology. Restating this, the framework must provide guidance for today's projects and build a bridge to the technology of the next decade.

Observations on Software Engineering

My area of interest is software engineering, which—in a narrow sense—involves the managing, conducting and evaluating the development and maintenance of software components. It is a branch of engineering in that it focuses on the creation of useful artifacts through the application of scientific principles. It differs from most other engineering disciplines in that software engineering is not bound by the physical laws of nature; rather it is guided by the models that formalize our current understanding. That is, unlike electrical engineering, which must respond to repeatable, external phenomena, the solution space of the software engineer is dominated by the formal models created by computer science (e.g., programming languages, tools for representing abstractions). The software engineer's models are artifacts: products of human creativity. Unlike the laws that explain the behavior of electrons, these models establish an approach to software development that becomes a self-fulfilling prophecy. The software engineer's interpretation of the problem determines his response to it, and one theme of my research is that the software engineer begins with an imperfect problem statement.

Although software engineers focus on a software product, it must be recognized that virtually every software product must operate on a computer (i.e., with hardware) and interact with users (i.e., humans). Thus, software is always a part of some larger system (which, in turn, may be part of an even larger system). Moreover, the goal of that system is to meet some need in the application domain. The interpretation of a software product outside the scope of the system and the need it addresses represents a level of abstraction fraught with danger. Thus, the primary challenge that the software engineer faces is not how to write or modify some piece of code; rather, it is to understand how that code meets some need. The software engineer must recognize that he is performing systems engineering and that the code is simply an expression of the most detailed design of a response to a need. Unfortunately, this mode of thinking is not very common, which can result in long-term consequences over the life of a system.

The seeds of today's software orientation were sowed in the early days of computing. The first need was to produce programs; symbolic assemblers and high-order languages made that task easier. Once we mastered the writing of programs, we confronted the difficulty in creating systems. The discipline of software engineering was spawned by the NATO-sponsored conferences [NaRa69]. These meetings focused on the development of large-scale, system-oriented software. The waterfall flow, first introduced by Royce [Royc70] in 1970 and later refined by Boehm [Boeh76], introduced a phased development. One could not go on to the next phase until the previous phase was complete and validated. The output from each phase was used to define the scope of its successor phase, and the model provided for corrective feedback to earlier phases. This model for software development was copied from experience with hardware; indeed, Boehm's waterfall diagram

differed from the hardware flow only in the use of "software" in the title "Software Requirements" and the relabeling of "Fabrication" "Code and Debug."

This parallel between software and hardware continues today.* Here is how a recent book puts it [CaG190].

Software design is the product engineering part of software development.
Programming is in some ways analogous to the manufacturing part.

Although the waterfall diagram has been much maligned, most of the proposed alternatives can be seen as adaptations of that basic model. Prototyping was introduced as a means to validate the specification before the phased development begins [GoSc81]. The spiral model emphasizes the risk-reduction activities in the early phases of development [Boeh88]. It too creates a valid specification, and implementation follows a traditional phased approach. Incremental development divides the process into layered builds, with each build following a phased plan [Gilb88]. In each of these models, the software process begins with a definition of what the system should do. This is followed by a design of how the system should be implemented, and—once the design is detailed enough to permit coding—the programs are implemented. Programs are tested, and tested components are then integrated and tested again. The perception is that software, like hardware, is a product to be implemented. Analysis and design determine what the product should do and how it should be constructed; integration and test establish that it performs as desired.

I believe that this hardware-based, product-oriented view of the software process has led us to focus on the software implementation rather than the knowledge that motivated its creation [Blum92c]. This idea can best be introduced by way of the software process metamodel in Figure 1. It presents the essence of the software process as a transformation from some need in an application domain into a software implementation that responds to that need. Two nonintersecting modeling lines are shown. The *conceptual models* reflect the application domain perspective; they describe the *proposed response to the need*. Although the conceptual models use domain formalisms and express the domain specialists' intent, they are not formal in the computer science sense. They are termed conceptual because they *describe* but do not *prescribe* the software solution. The conceptual models must be transformed into *formal models* that establish the essential behaviors and performance of the *desired software product*. Finally, details are added until an implementation exists that is correct with respect to the formal model. The implementation, of course, is also a formal model.

Software, however, is not static. Lehman defines E-type programs as programs that alter the requirements to which they respond, thereby initiating a demand for change [Lehm80]. Thus, the metamodel of Figure 1 is but one iteration within a continuing cycle of change and improvement. The transformation represented by this model, from a need to a software product intended to meet that need, can be decomposed into a sequence of

* Although there are some software process models, such as the operational approach [Zave84] and model execution [Hare92], that do not echo the hardware development model, space does not allow us to consider them here. For a more complete discussion of this topic see [Blum92a, Blum92b].

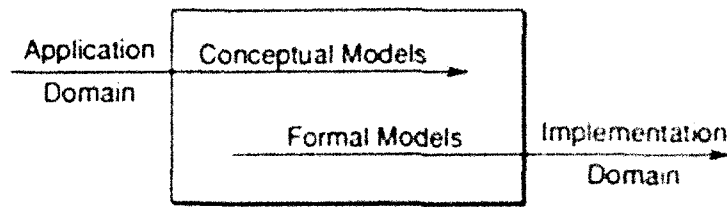


Figure 1 The essential software process.

three transformations.

From the need identified in the application domain to the conceptual model that establishes how the technology can provide an appropriate solution. Here analysts require a deep understanding of the application domain plus knowledge of the potential solutions supported by the technology.

From the conceptual model, which describes in terms natural to the domain specialist what is to be implemented, to a formal model, which establishes the behavior and performance of the product to be delivered.

From the formal model to the implementation. This is the historic domain of software engineering. The final step in this process (e.g., compilation) is always automated.

Notice that many distinct classes of conceptual model will be valid responses to a given need, many distinct classes of formal model will be valid translations of a given conceptual model, and many distinct classes of implementation will be correct for a given formal model. Thus, the software process (and, indeed, every design process) involves successive restrictions of the solution space until only one solution exists. Knowledge related to rejected alternatives seldom is retained. This implies that, as the product evolves, we enrich our understanding of the particular solution that the product represents, but we lose knowledge associated only with alternative solutions.

It is important to distinguish between a need and a specific response to that need. As it is currently constituted, the software process narrows the solution space to realize a particular response to a need. The need, however, can be open (i.e., there are many potential responses that could satisfy it) or closed (i.e., the solution implies a specific response) [BIMo92]. Consequently, there will be many conceptual and formal models that can fulfill the intent of an open application need, and, in contrast, few models will exist that can satisfy the requirements of a closed objective. But once a solution is accepted and expressed as a formal model, the problem space is changed. Correctness of the software is with respect to that particular solution (i.e., not with respect to the initial need). After the formal model exists, the design is open only to the extent that alternative designs are available to satisfy the requirements (i.e., the problem space becomes that of software implementation). When one begins with an understanding of the application need, the openness of the problem is apparent. On the other hand, if one begins with the existing implementation, it is very difficult to distinguish between the results of a design decision and

the inherent constraints of the problem. That is, when going from the left to the right in Figure 1, one sees the software in the context of the problem to be solved. In contrast, when going from the right to the left, one sees the problem in the context of a particular response to it.

This difference in perspectives is captured in Figure 2. It shows two software paradigms. The *product-oriented* paradigm is the traditional hardware-based view of the process. One begins with a system specification (i.e., the formal model of Figure 1) and concludes with the implementation of the system. The *problem-oriented* paradigm, on the other hand, operates within the problem space, it begins with the identification of a solution to the problem and ends with the complete design of that solution. In the context of the three transformations in Figure 1, the product orientation focuses on the third transformation and the problem orientation on the first two. (The solution design is the detailed formal model.) For closed problems, there are few solutions, and one can be specified and implemented. For open problems, however, there is the danger that the product will be bound to a solution that becomes obsolescent; here it is desirable to retain a full understanding of the problem to be solved and the alternative solutions under consideration. Unfortunately, the two paradigms are incompatible, they cannot be merged, and one cannot evolve from the other. And this places us on the horns of a dilemma. Many of the Navy tactical systems involve hardware, which must be specified before it is manufactured, yet—as noted in the introduction—most improvements to Navy tactical systems will come through incremental enhancements.

	Product Oriented	Problem Oriented
Top	System Specification	Solution Identification
Bottom	System Implementation	Solution Design

Figure 2 Two software paradigms.

The consequences of this tension between system goals and product structure are illustrated in the next three figures. Figure 3 depicts the formal knowledge of a system developed with a product-oriented paradigm. It depicts a concept of operations, which is supported by a requirements document that defines a specific system to support that concept of operations. The remaining four trapezoids in the diagram represent the successive levels of detail necessary to produce a product that supports the intended concept of operations in the prescribed manner. (The diagram shows the source code as the lowest level of design detail, not a product created from the design.) If the concept of operations is static and if the knowledge is well documented, then the knowledge structure shown in Figure 3 is very effective. One can trace code to requirements and concepts to operational entities. But there are two fundamental problems with the knowledge base in Figure 3. First, the knowledge is poorly organized, often incomplete, and difficult to access

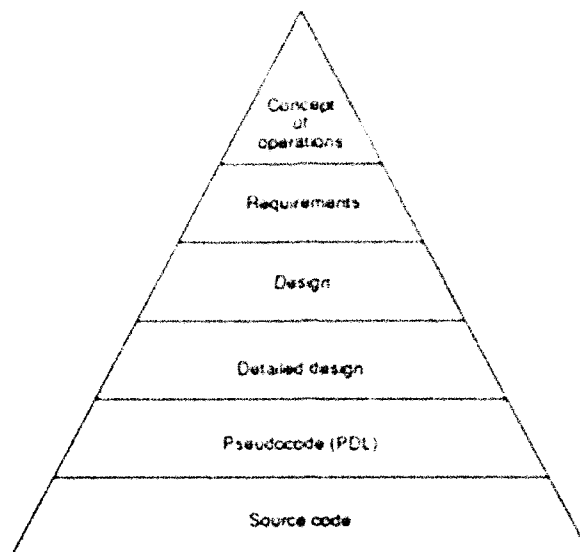


Figure 3 Knowledge of a new system.

or integrate. It tends to be document based, and there are relatively few formally maintained links between levels (e.g., changes to the source code may not be reflected in the PDL and vice versa). Second, the knowledge base is dynamic, and the concept of operations shifts as system experience grows and as the external environment changes.

Figure 4 demonstrates the knowledge shift as a concept of operations adapts to external requirements. In this example, the concept of operations has identified a completely new mission, but the product itself has not changed (i.e., the source code is unaltered). Now there is a mismatch between what the program (source code) does and what is needed (the concept of operations). The figure shows the requirements for a new product that will respond to the new need. Although the new requirements differ from the old requirements, much of the design (and source code) can be reused. Dashed lines depict the parts of the existing system that are obsolescent with respect to the new concept of operations. In an era of new system development, the "safe" approach would be to scrap the old system and custom build a new system. Such an attack is now recognized as being too expensive and of too high a risk. Consequently, there is a need to reengineer reusable components in the older system and to guide the development of the new system in the exploitation of the reusable components [Free87, Trac88, PrAr91, Boeh90]. From a knowledge-based perspective, the challenge is to look down from the concept of operations to identify what concepts and operations can be generalized (e.g., the domain analysis orientation) and look up from the level of the source code to identify what components can be reengineered for reuse (e.g., the component library orientation). Unfortunately, the knowledge, arranged so neatly in Figure 3, is not structured to support such a transition.

A different change scenario is presented in Figure 5. Here the concept of operations document has remained relatively fixed, but the programs have been altered. Thus, although there have been many changes to the source code, the rationale for these changes may not have been documented as changes to the concept of operations or the

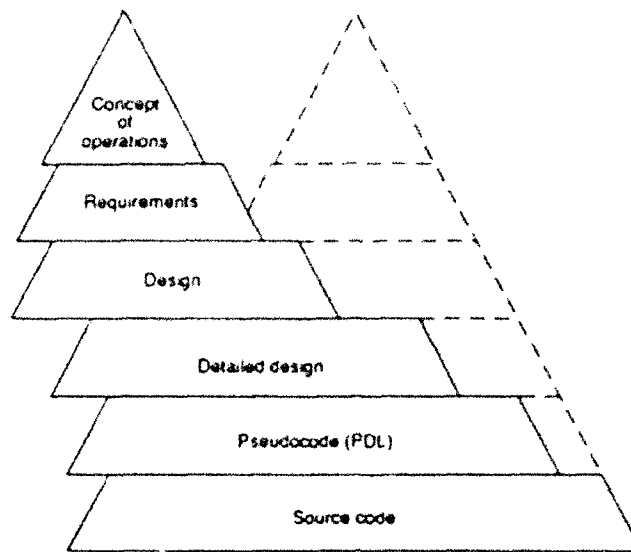


Figure 4 Knowledge of an existing system and a new concept of operations.

requirements. Because the higher level documentation (i.e., knowledge) is out of date, it is viewed as untrustworthy, and the incentive for not updating the higher level documents increases. That is, because the documentation has not been updated, it is not used; because it is not used, it will not be updated. Confronted with this reality, the maintainer focuses on the *object of change* and not the *reason for change*. (That is, on the product to be altered and not the problem to be solved.) Consequently, changes are limited to what can be understood, and knowledge of the system degrades further.

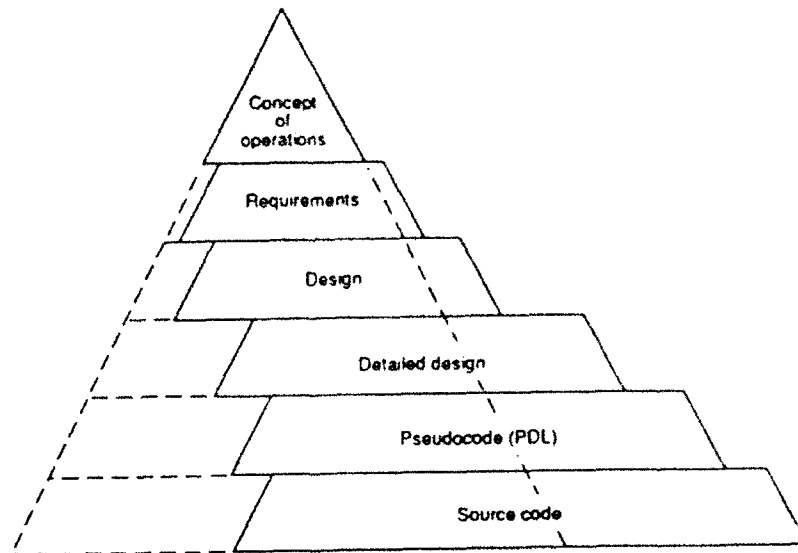


Figure 5 Knowledge of a system as it evolves.

One method for relieving this tension is to institute a knowledge-based approach in which knowledge of the application domain, the technology used, and specific systems are maintained in an integrated manner that permits reuse, prototyping, and the partially automated generation of products. There are many researchers working on responses of this general category, and the purpose of this paper is to establish a framework for classifying Navy activities so that the experience may be exploited in the development and adoption of new paradigms. In the context of what has been presented in this section, my personal view is that we are in a product-oriented paradigm and should move to a problem-oriented paradigm. Many (including myself) are working on alternative approaches, and it would be premature to speculate about the form of systems in that paradigm. Nevertheless, it should be clear that the new paradigm will be knowledge-based, that it will offer little immediate help to projects embedded in a product-oriented paradigm, and that it represents our best hope for the inexpensive, reliable, and flexible evolution of future Navy tactical systems.

A Engineering/Reengineering Framework

The objective of this section is to identify a framework for the description, collection, and assessment of engineering/reengineering projects for complex systems. Lacking such a framework, commonality among projects will be obscured, and the transfer of project experience will be degraded. The framework should have two important properties.

It should aid the project organizers and evaluators in the articulation of the project goals.

It should aid in the refinement of cross-project knowledge and facilitate the introduction of new concepts and methods.

At this point the definition of the framework is speculative, and it has not been tested with the classification of real projects. To facilitate analysis at this early stage of analysis, I restrict the framework definition to just the engineering/reengineering of the software components in a system. The framework, however, should be extensible to include all system components.

There are four dimensions in the software engineering/reengineering framework.

Problem granularity characterizes the problem to be solved by the project. It may range from a full system to be engineered to a single component to be re-engineered. Associated with granularity are effort, cost, and schedule constraints plus estimates of the available experience. The objective is to associate experience with some granularity measure so that, for example, experience with 10 effort-year projects can be referenced by other projects of comparable size.

Problem level characterizes the level of problem addressed. I identify three levels.

Knowledge oriented. This is the level just described as problem-oriented. Its goal is to use knowledge to guide aspects of the process. Examples include domain analysis, megaprogramming, and the operational approach.

Object oriented. Although this is a product-oriented view, it is higher level than that of the code. It maximizes the benefit provided by encapsulation and information hiding. Projects that employ the Ada programming language at the design level provide experience at this level.

Product oriented. This is the lowest-level of engineering/reengineering. Its goal is to produce components, and the emphasis is on the component, not the domain activity it supports. Routine program maintenance is an illustration of a project at this level.

The goal of software engineering should be to raise the level of the problem being addressed.

Project motivation classifies the rationale behind the project's initiation. I identify five motivations.

New product. This is the creation of a new product.

Correction. This is a modification of an existing product to correct a fault or deficiency. It is similar to the error repair of corrective maintenance.

Adaptation. This is the modification of an existing product that does not alter the functionality of the product but that alters the product to accommodate an altered environment (e.g., changes to a fire-control software module to conform to an altered radar interface).

Enhancement. This is the modification of an existing product to alter and improve its functionality, performance, etc. It is similar to perfective maintenance.

Experimental. This motivation is reserved for projects that experiment with a new technology or concept (e.g., the conversion of CSM-2 programs to Ada).

Some projects may have more than one motivation, but the framework may permit only one motivation for a project.

Supporting paradigm refers to the problem-oriented and product-oriented paradigms depicted in Figure 2. For the near term, the definition of the problem-oriented paradigm is extended to include both projects that employ that paradigm and projects that have, as a primary goal, the building of a knowledge base for potential use by a method employing the problem-oriented paradigm.

The benefit of this framework is that all projects can be placed in a project space that permits comparisons of characteristics across projects. For example, lines of code per hour measures are like blood pressure readings; without knowing the patient, diagnosis, and therapy, a reading of 150/85 is meaningless. By tying a project to a node in the framework, one has a baseline for comparing results from different projects or methods. One also can use the framework to organize more detailed investigations. For instance, consider the problem level dimension for reengineering projects.

Reengineering at the knowledge-oriented level treats the knowledge in the system documents as a unified whole that permits integration of concepts, reuse of evaluation tools and techniques, and access to both current and historical design information. Therefore, the framework at this level can consider questions such as:

What knowledge, documentation, simulations, and other tools are available for systems, and how accurate, flexible, and transportable are they?

What knowledge is available in forms that can be processed and indexed within an integrated database? What is the granularity of this knowledge, and can it be adapted for processing by off-the-shelf tools?

What research in knowledge representation, simulation, model execution, and so on, would be applicable to adaptation in support of reengineering?

At the knowledge-oriented level, the intent is to understand the domain so that a transition plan (or bridge to the new technology) can be proposed. For the object-oriented level, the goal is to evaluate the success in utilizing process-improvement techniques that emphasize encapsulation, reuse, and components. In the forward engineering view, the following questions can be answered:

How are new development activities exploiting the features of Ada? Are there project evaluations that would aid other projects? Are there libraries available for exchange?

Are there measures for the degree of encapsulation and reuse employed? Are there revised models for the software process using these development methods? How reliable are these measures?

For reengineering additional questions can be addressed:

To what extent has reengineering used an object-oriented level of abstraction? What are the costs and benefits of this technique?

How is encapsulation employed in Ada-based reengineered software projects? What is reused and what is packaged for reuse by subsequent packages? Which of the Ada features are used in reengineered software? How (if at all) is the documentation altered to accommodate the object orientation?

At the product-oriented level, the primary concern is for the methods used in transforming a product from one form (e.g., code in CSM-2) to another (e.g., code in Ada). Of particular interest here are questions such as:

What criteria were used to decide to reengineer a component? To control and evaluate the reengineering activity? To validate the reengineered product?

What technology and tools were used to reengineer the product? How was the design knowledge captured? What level of abstraction was used to guide the forward engineering process?

What was the motivation for the reengineering (e.g., improved interoperability, hardware change, altered functionality)? Are there different reengineering methods for different motivations?

Thus, the availability of a sound framework not only guides in the analysis of project data, but it also assists in the definition of new analytic efforts.

Summary

This paper began with an examination of Navy tactical systems and observed that most future improvements will result from the evolution of existing systems rather than the development of new systems. The paper then addressed the engineering/reengineering issues associated with this need for continuing evolution. The emphasis was placed on the software process, and the discussion concluded that (a) the present paradigm was limited, (b) no near-term alternatives are available for complex Navy projects, and (c) there is enough advanced knowledge to support the building of bridges from the present to the future paradigms. An organizing framework was introduced that can guide in the analysis of data and assist in the formulation of new studies.

References

- [BlMo92] Blum, B. I. and T. Moore, Representing Navy Tactical Computer System Knowledge for Reengineering and Integration, *2nd International Conf. on Systems Integration*, (in press).
- [Blum90] B. I. Blum, *TEDIUM and the Software Process*, MIT Press, Cambridge, MA, 1990.
- [Blum92a] B. I. Blum, *Software Engineering: A Holistic View*, Oxford University Press, New York, NY, 1992.
- [Blum92b] B. I. Blum, The Economics of Adaptive Design, *J. of Systems and Software*, (in press).

- [Blum92c] B. I. Blum, On the Development of Hardware and Software, *Information and Decision Technology*, (in press).
- [Boeh76] B. W. Boehm, Software Engineering, *IEEE Transactions on Computers*, C-25:1226-1241, 1976.
- [Boeh88] B. W. Boehm, A Spiral Model of Software Development and Enhancement, *Computer*, (21,5):61-72, 1988.
- [Boeh90] B. W. Boehm, DARPA Software Strategic Plan, *Proc. ISTO Software Technology Community Meeting*, June 27-29, 1990.
- [CaGl90] D. N. Card with R. L. Glass, *Measuring Software Design Quality*, Prentice Hall, Englewood Cliffs, NJ, 1990.
- [EnLB89] J. P. Enterline, R. E. Lenhard and B. I. Blum (eds.), *A Clinical Information System for Oncology*, Springer-Verlag, New York, 1989.
- [Free87] P. Freeman, *Tutorial on Software Reusability*, IEEE Computer Society Press, Washington, DC, 1987.
- [Gilb88] T. Gilb, *Principles of Software Engineering Management*, Addison Wesley, Reading, MA, 1988.
- [GoSc81] H. Goma and D. B. H. Scott, Prototyping as a Tool in the Specification of User Requirements, *Proceedings, International Conference on Software Engineering*, pp.333-342, 1981.
- [Hare92] D. Harel, Biting the Silver Bullet, *Computer*, pp. 8-20, January, 1992.
- [Lehm80] M. M. Lehman, Life Cycles and Laws of Program Evolution, *Proc. IEEE*, 68:1060-1076, 1980.
- [NaRa69] P. Naur and B. Randell (eds.), *Software Engineering: Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*, Scientific Affairs Division, NATO, Brussels, 1969.
- [PrAr91] R. Prieto-Diaz and G. Arango, *Tutorial: Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press, Silver Spring, MD, 1991.
- [Royc70] W. W. Royce, Managing the Development of Large Software Systems, *IEEE WESCON*, pp. 1-9, 1970.
- [Trac88] W. Tracz, *Software Reuse -- Emerging Technologies*, IEEE Computer Society, 1988.
- [Zave84] P. Zave, The Operational Versus the Conventional Approach to Software Development, *Communication of the ACM*, 27:104-118, 1984.

A View to an Implementation

Ngocdung T. Hoang
Naval Surface Warfare Center
Silver Spring, MD 20903-5000

Nicholas Karangelen
Trident Systems Incorporated
Fairfax, VA 22030

ABSTRACT

The representation of resources is a major issue in designing large and complex systems. The ability to represent and analyze these resources early in the design process supports the understanding of how resources are utilized, resulting in a major cost reduction in system integration. This paper presents a method to generate the system's Implementation Capture View. This view is defined as a documentation of the resources and their interfaces which make up the system under design including the hardware, software, and human operators. The Implementation Capture View also includes documentation of the resource selection and design rationale, and a mapping from the Functional and Behavioral Capture Views to the resources in the Implementation Capture View.

Introduction

Research on the implementation part of the system has been conducted and is being continuously updated. Mostly the implementation issue is addressed as part of the design process, and the representation of the system's resources is limited to a certain design scope. In their book [Rum], Rumbaugh, et al., present an Object-Oriented based approach in dealing with the implementation issue, but the method only emphasizes the problem of small-to-medium sized and software-oriented systems. The same experience was found in the Structured Analysis method [You].

The design of large, mission critical systems demands an understanding of all system characteristics. The concept of using the five system capture views (Informational, Functional, Behavioral, Implementation, and Environmental) to completely represent the system has been introduced [Hoa]. Though these system capture views were identified, much work will be needed to specify each of their capturing formats. Also, the relationship between these views must be identified such that the completeness of the design capture is fulfilled. The focus of this paper is on the Implementation Capture View and the relationship between this view and the Functional and Behavioral Capture View.

The Implementation Capture View documents the architectural descriptions and performance capabilities of all hardware, software, and human resources which represent a particular embodiment of the system under design. The hardware architecture describes the physical resources of the system including the components, interconnection topology and protocol. The software architecture describes the Computer Software Configuration Items (CSCI) and the executable software tasks including the messages passed between modules. Finally, the Humware Architecture describes the number of personnel required to operate the system under various conditions and the level of training and experience for each operator. The hardware, software and humware architectures are captured in a database which also includes the resource selection, design rationale, and the traceability of system requirements through the design.

The description of the resource architectures represents the principal products of the systems engineering effort and establishes a baseline for the detailed design of the system and operating concept. The architecture descriptions provide a basis for the development of system specifications (e.g., DOD-STD-2167A System Segment Specification (SSS), Software Requirements Specification (SRS), etc.). These specifications in turn provide the project software and hardware engineering teams with the basis for detailed system design. They also establish a basis for development and analysis of a performance simulation for the system under design. Simulation provides the ability to identify potential shortfalls and errors in the system design early in the design cycle when changes and corrections are considerably less costly.

Although the capturing technique is described as a series of steps or activities which are presented in a particular order, this sequence is not intended to be a rigid formula for generating the Implementation Capture View. The order of the steps represents a general flow of activity which is intended to be iterative both between steps and across the overall process. The following sections describe the information which must be captured and a preliminary process for accomplishing that capture. It is important to note that this methodology only emphasizes the generation of the Implementation Capture View based on a predefined Functional and Behavioral Capture View.

Identify System/Subsystem Function of Interest

The first step in developing an Implementation Capture View is to identify a complete logical model of the system at some level of decomposition or design level of detail. Sub-systems or components of a larger system can also be addressed; however, it is important to clearly define the boundaries of the logical model to be implemented. This becomes particularly important when the logical and implementation models of the system (or sub-system) under design are simulated. Simulation of a given design requires explicit external interface definitions and an unambiguous definition of the sim/stim requirements.

This step is a precursor to developing a function-resource mapping in that it defines a consistent and complete function listing for the system under design at a given level of detail. The mapping may be accomplished at any level of functional decomposition; however, the functions mapped must represent the entire system (or clearly defined sub-system). High or low levels of abstraction (or some combination) may be used, however, no redundant function capture is allowed (i.e., function and its children). An example of identifying a complete logical model at a given level is illustrated in Figure 1. This example shows the example sonar problem functional decomposition down to three levels from the context diagram.

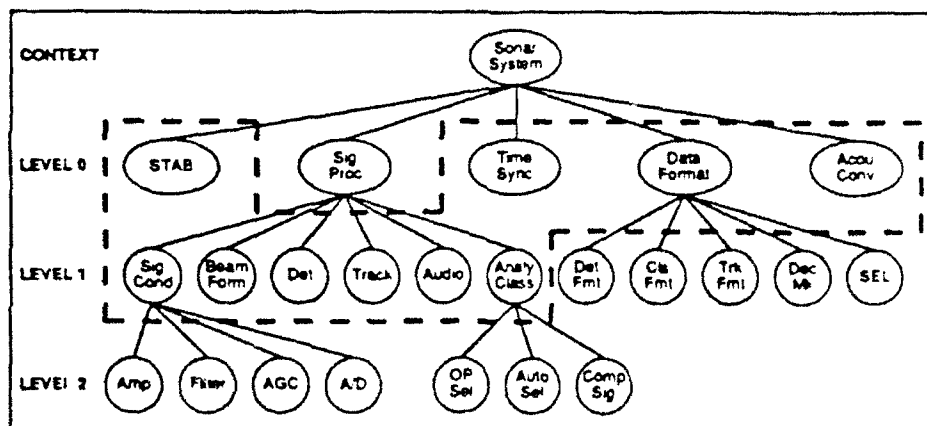


Figure 1. Example Sonar System Functional Decomposition

The solid lines connecting the functions represent the parent/child relationships in the decomposition. The dashed line encloses a set of functions which represent the entire system. Note that functions at two different levels are used to represent the system, but no overlap (i.e., inclusion of a function's parent or child) is allowed. Any overlap in this selection of functions to represent the system would result in duplication and ambiguity in the associated data flow model.

Resource Library

A resource library is a repository for all candidate resources that can be used or built into the design process. The resource library exists in the form of a database where all candidate resources are categorized from a very general class to a specific type. While the descriptions and performance capabilities of the off-the-shelf products are documented according to their manufacturing specifications, the "to-be-designed" or modified resources are listed by their expected values including their design constraint. The multi-level resource classification allows the early prediction of system performance without constraining the design to a specific type of resource.

The resource library includes both "black box" resources and complex resources. Black box resources are described in terms of their physical characteristic, performance, and other design factors but are not decomposed into component parts. This does not imply that black box resources are simple, only that their components are not described in the resource library. Complex resources are also described in terms of selected design factors but, in addition, include a component level description which identifies the constituent parts of the complex resource and the internal interconnection of the parts. Cross references are provided for component parts and related sub-systems which are also found elsewhere in the resource library. A graphical user interface which provides point and click access to the resource descriptions contained in the library is envisioned to facilitate ease of use.

The resource library can be used as a reference point for developing the system resource architectures and must be established before the function-resource mapping process. Figure 2 illustrates an example hierarchy for organizing the resource library used in developing the passive sonar system example.

Resource Description

Regardless of its type or class, each resource description must be documented in detail. An appropriate format for capturing each type of resource should be generated to formalize the representation. The format contains a number of fields which describe the resource in terms of selected design factors, interfaces, and components (if provided). Multiple fields are provided for each design factor allowing specified values, measured values, etc., to be entered and maintained. Design rationales and alternative resources are also listed for alternate designs. Figure 3 illustrates the description of an example format for a general purpose CPU type resource.

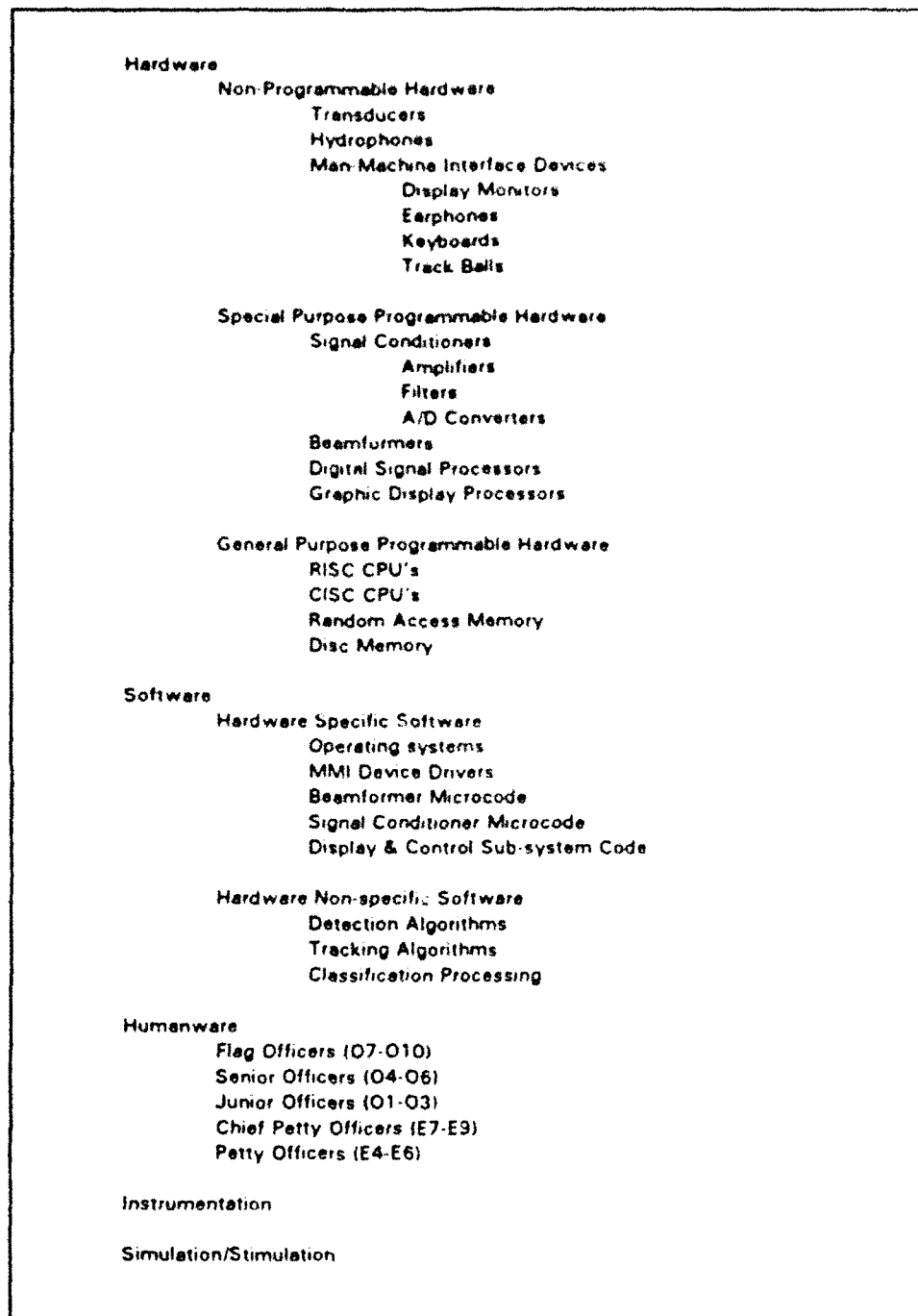


Figure 2. Example of a Resource Library Organization

Resource Name/Title ONE MIP CPU	Availability: TBD Inherent Availability: TBD Achieved Availability: TBD Operational Availability: TBD Ease of Replacement: TBD Crash Recoverability: TBD Computation Heavy Process Effects: TBD 4.5 Quality: TBD
Resource Type General Purpose Computer Hardware	5. Security 5.1 Level: Unless
Resource Description Provides general purpose computing capability including CISC processor with floating point numeric coprocessor.	Components TBS
Resource Selection Rationale TBS	Interfaces
Design Factors	Serial Ports
1. Performance	Type (e.g. RS232, IEEE 488, etc.)
1.1 Response Time: TBD	Speed (e.g. 19.2 KB, 9.6 KB, etc.)
1.2 Capability:	Parallel Ports
1.2.1 CPU Processor Type (e.g. 68040, 80486, etc.)	Type (e.g. SCSI, Centronics, etc.)
1.2.2 Bus Architecture (e.g. VME, Future bus, etc.)	No. of Signal Lines: TBD
1.2.3 CPU Word Size: 32 bits	Speed: TBD
1.2.4 Address Bus Size: TBS	Network Ports
1.2.5 Data Bus Size: TBS	Type (e.g. Ethernet, FDDI, etc.)
1.3 Relative Activity: TBD	Connector Type (Fibre optic, copper)
1.4 Speed:	Speed (e.g. 10 MBPS, 100 MBPS)
1.4.1 CPU Clock speed: TBS	DMA Ports:
1.4.2 MIPS: one million 32 bit inst per second	Type (e.g. DRV11W, etc.)
1.4.3 FLOPS: TBS	No. of Signal Lines: TBD
1.5 Throughput: TBD	Speed: TBD
1.6 Latency: N/A	Bus Connections
1.7 Efficiency: N/A	Type
1.8 Predictability: TBD	Speed
2. Real-Time	Installed Location
2.1 Deadlines	Eight ONE MIP CPU's are installed in the system.
2.1.1 Hard Deadline: N/A	CPU's 1 thru 4 are connected to Ethernet Network #1 and CPU's 5 thru 8 are connected to Ethernet Network #2
2.1.2 Soft Deadline: N/A	
2.2 Synchronization: TBD	
3. Computation/Processing Requirements	
3.1 Importance: TBD	
3.2 Usefulness: TBD	
3.3 Priority: TBD	
3.4 (Computing) Portability: TBD	
3.5 Interrupt/Reset Capabilities: TBD	
3.6 Memory Space:	
3.6.1 Main Memory Installed: TBS	
3.6.1 Main Memory Max Addressable: TBS	
3.6.2 CPU Cache Memory Size: TBS	
3.6.2 Disk Memory Max Addressable: TBS	
4. Dependability	
4.1 Reliability: TBD	
4.2 Accuracy: TBD	
4.3 Fault Tolerance: TBD	
4.4 Availability	

Figure 3. Example format for a General Purpose One MIP CPU

Multi-level Function-Resource Mapping

The multi-level function-resource mapping provides a mechanism for allocating functions to resource types and ultimately for mapping those functions indirectly to specific resources. The mapping establishes a strong link between the logical architecture and the resource architecture and requires that (1) every function be fully implemented in resources, and (2) every resource be traceable back to a required function (or a derived system service task).

Once a complete functional listing is established and a preliminary resource library exists, the function-resource mapping can be performed in various layers or steps. The function-resource mapping process is intended to be iterative and as such must accommodate numerous changes. Although in the following discussion the mapping is shown in three layers, it can vary depending on how much trade-off systems engineers would like to estimate. Opportunities exist at each step or layer to optimize the mapping according to a variety of design factors. However, increasing the number of layer of the mapping will effect the development cost; hence, system engineers should have an appropriate mapping plan to suit their requirement. At each level of mapping, analysis techniques can be applied to verify the correctness of the design. Figure 4. illustrates the three level function-resource mapping of a passive sonar system.

The first layer of the function-resource mapping is an assignment of the system functions to generalized resource classes. Each function is mapped to one of these generalized resource classes which is then further refined by specifying a certain resource type from that class. For example, the beamforming function is mapped to the general class of Programmable Hardware & Software, which is further specified as special purpose custom beamformer hardware with beamformer microcode software.

The second layer of the function-resource mapping establishes a set of implementation tasks which will be performed by the specific resource classes or types. Many of these implementation tasks are directly related to the functions and may represent an implementation specific functional partitioning, grouping or some combination of the two which reflects the intended implementation. Other implementation tasks are created to provide required implementation specific system services such as CPU operating systems, database managers and network executives. This layer of the mapping provides the systems engineer with a mechanism for repartitioning the functional decomposition for implementation without directly modifying the Functional Capture View.

The third layer of function-resource mapping is the allocation of the implementation tasks to specific resources. At this level the description of the candidate resources are detailed enough such that a framework for the system design can be constructed. In this layer of the mapping individual resources are identified by hardware unit number, software/humware task and human operator for each implementation task. Many implementation tasks require both hardware and software resources. The static allocation of implementation tasks to specific combinations of hardware and software represents an over simplification of the system's hardware-software mapping. In a system with alternate program load options or in systems where software tasks are dynamically allocated to hardware this level of the mapping simply represents a particular instance of the possible software hardware mapping.

First Level		Second Level		Third Level		SW Task Reassignment Competition Hardware Resource Types	Priority Resource (0-5)
Function	General Resource Class	Specific Resource Type	Task ID	Task Description	Task Allocation HW Unit or Operator	SW Task or Process	
Acoustic Energy Core	Non Prog HW	Hyd Array	HW Task 101	Acoustic Energy Core	Hyd Array	N/A	N/A
Signal Conditioning	Prog HW & SW	Custom Signal Core SC Microcode/CSC1.1	HW/SW Task 201	Sig Core Hydr 1-600	SC #1	SW Task 201	Not relocatable
			HW/SW Task 202	Sig Core Hydr 601-800	SC #2	SW Task 202	Not relocatable
			HW/SW Task 203	Sig Core Hydr 801-1200	SC #3	SW Task 203	Not relocatable
			HW/SW Task 204	Sig Core Hydr 1201-1600	SC #4	SW Task 204	Not relocatable
Beamforming	Prog HW & SW	Custom Beamformer BF Microcode/CSC1.2	HW/SW Task 305	Form Fixed Del Beams	BF #1	SW Task 305	BF #1 or BF #2
			HW/SW Task 306	Form Steerable Beams	BF #2	SW Task 306	BF #1 or BF #2
Detection	Prog HW & SW	One MP CPU Ada Code/CSC1.3	HW/SW Task 101	Det Beams 1-20	CPU #1	SW Task 101	CPU #1 thru CPU #6
			HW/SW Task 102	Det Beams 21-40	CPU #1	SW Task 102	CPU #1 thru CPU #6
			HW/SW Task 103	Det Beams 41-60	CPU #1	SW Task 103	CPU #1 thru CPU #6
			HW/SW Task 104	Det Beams 61-80	CPU #2	SW Task 104	CPU #1 thru CPU #6
			HW/SW Task 105	Det Beams 80-100	CPU #2	SW Task 105	CPU #1 thru CPU #6
			HW/SW Task 106	Det Beams 101-120	CPU #2	SW Task 106	CPU #1 thru CPU #6
Tracking	Prog HW & SW	One MP CPU Ada Code/CSC1.4	HW/SW Task 111	Trk Beams 1-20	CPU #3	SW Task 111	CPU #1 thru CPU #6
			HW/SW Task 112	Trk Beams 21-40	CPU #3	SW Task 112	CPU #1 thru CPU #6
			HW/SW Task 113	Trk Beams 41-60	CPU #3	SW Task 113	CPU #1 thru CPU #6
			HW/SW Task 114	Trk Beams 61-80	CPU #4	SW Task 114	CPU #1 thru CPU #6
			HW/SW Task 115	Trk Beams 81-100	CPU #4	SW Task 115	CPU #1 thru CPU #6
			HW/SW Task 116	Trk Beams 101-120	CPU #4	SW Task 116	CPU #1 thru CPU #6
Analysis & Classification	Prog HW & SW	One MP CPU Ada Code/CSC1.5	HW/SW Task 121	Analysis Ch 1-4	CPU #5	SW Task 121	CPU #1 thru CPU #6
			HW/SW Task 122	Analysis Ch 5-8	CPU #5	SW Task 122	CPU #1 thru CPU #6
			HW/SW Task 123	Analysis Ch 9-12	CPU #7	SW Task 123	CPU #1 thru CPU #6
			HW/SW Task 124	Analysis Mgmt	CPU #8	SW Task 124	CPU #1 thru CPU #6
Audio	Prog HW & SW	One MP CPU Ada Code/CSC1.7	HW/SW Task 131	Operator #1 Audio	DC#1	SW Task 131	DC #1 thru DC #4
			HW/SW Task 132	Operator #2 Audio	DC#2	SW Task 132	DC #1 thru DC #4
			HW/SW Task 133	Operator #3 Audio	DC#3	SW Task 133	DC #1 thru DC #4
			HW/SW Task 134	Operator #4 Audio	DC#4	SW Task 134	DC #1 thru DC #4
Stabilization	Prog HW & SW	One MP CPU Ada Code/CSC1.8	HW/SW Task 141	Stabilization	CPU #6	SW Task 141	CPU #1 thru CPU #6
Time Sync	Prog HW & SW	One MP CPU Ada Code/CSC1.9	HW/SW Task 151	Time Sync	CPU #6	SW Task 151	CPU #1 thru CPU #6
Data Formatting & Option Selection	Prog HW & SW	Display Console Ada Code/CSC1.10	HW/SW Task 161	Detection C&D	DC#1	SW Task 161	DC #1 thru DC #4
			HW/SW Task 162	Track C&D	DC#2	SW Task 162	DC #1 thru DC #4
			HW/SW Task 163	Analysis C&D	DC#3	SW Task 163	DC #1 thru DC #4
			HW/SW Task 164	Maintenance C&D	DC#4	SW Task 164	DC #1 thru DC #4
	Human	BT 3	Hum Task 001	Detection Operator	OP #1	Hum Task 01	OP #1 thru OP #4
			Hum Task 002	Track Operator	OP #2	Hum Task 02	OP #1 thru OP #4
			Hum Task 003	Analysis Operator	OP #3	Hum Task 03	OP #1 thru OP #4
			Hum Task 004	Maintenance Operator	OP #4	Hum Task 04	OP #1 thru OP #4
	Prog HW & SW	One MP CPU Ada Code/CSC1.11	HW/SW Task 001	CPU #1 Op System	CPU #1	SW Task 001	Not relocatable
			HW/SW Task 002	CPU #2 Op System	CPU #2	SW Task 002	Not relocatable
			HW/SW Task 003	CPU #3 Op System	CPU #3	SW Task 003	Not relocatable
			HW/SW Task 004	CPU #4 Op System	CPU #4	SW Task 004	Not relocatable
			HW/SW Task 005	CPU #5 Op System	CPU #5	SW Task 005	Not relocatable
			HW/SW Task 006	CPU #6 Op System	CPU #6	SW Task 006	Not relocatable
			HW/SW Task 007	CPU #7 Op System	CPU #7	SW Task 007	Not relocatable
			HW/SW Task 008	CPU #8 Op System	CPU #8	SW Task 008	Not relocatable
			HW/SW Task 009	DC#1 Op System	DC#1	SW Task 009	Not relocatable
			HW/SW Task 010	DC#2 Op System	DC#2	SW Task 010	Not relocatable
			HW/SW Task 011	DC#3 Op System	DC#3	SW Task 011	Not relocatable
			HW/SW Task 012	DC#4 Op System	DC#4	SW Task 012	Not relocatable
			HW/SW Task 013	System Exec	CPU #1	SW Task 013	CPU #1 thru CPU #6
			HW/SW Task 171	CPU #1 Not Node	CPU #1	SW Task 171	Not relocatable
			HW/SW Task 172	CPU #2 Not Node	CPU #2	SW Task 172	Not relocatable
			HW/SW Task 173	CPU #3 Not Node	CPU #3	SW Task 173	Not relocatable
			HW/SW Task 174	CPU #4 Not Node	CPU #4	SW Task 174	Not relocatable
			HW/SW Task 175	CPU #5 Not Node	CPU #5	SW Task 175	Not relocatable
			HW/SW Task 176	CPU #6 Not Node	CPU #6	SW Task 176	Not relocatable
			HW/SW Task 177	CPU #7 Not Node	CPU #7	SW Task 177	Not relocatable
			HW/SW Task 178	CPU #8 Not Node	CPU #8	SW Task 178	Not relocatable
			HW/SW Task 179	DC#1 Not Node	DC#1	SW Task 179	Not relocatable
			HW/SW Task 180	DC#2 Not Node	DC#2	SW Task 180	Not relocatable
			HW/SW Task 181	DC#3 Not Node	DC#3	SW Task 181	Not relocatable
			HW/SW Task 182	DC#4 Not Node	DC#4	SW Task 182	Not relocatable
	Prog HW & SW	One MP CPU Ada Code/CSC1.13	HW/SW Task 181	CPU #1 DB Libr	CPU #1	SW Task 181	Not relocatable
			HW/SW Task 182	CPU #2 DB Libr	CPU #2	SW Task 182	Not relocatable

Figure 4. Three Levels of Function-Resource Mapping of the Passive Sonar Example

Resource Architecture

Resource architecture is the representation of system resources including their interconnection and utilization. With the multi-level resource description, the resource architectures can be constructed at any mapping level. The accuracy of the analysis is influenced by the abstraction level of the resource descriptions. When the mapping reaches certain levels of detail, the developed architectures will not only reflect the resources which will perform the required system's functions, but also the resources that provide the system support functions (i.e., operating system software). In general, each architecture represent the network of the same type of resources. In the computer-based system, three resource architectures, hardware, software, and humware are usually addressed. The following examples of the resource architectures reflect the third level of mapping that was mentioned above.

The hardware architecture includes a description of the hardware components of the system and their interconnection. Information about the hardware that was selected in the function-resource mapping is put together in the form of diagrams and a data base. The diagrams show the locations, components, and physical interconnection of the various hardware units. In addition to the fields provided in the resource library, the hardware architecture data base includes information on the selection rationale and requirements traceability for each hardware resource in the system, the installed location (both in physical terms and in terms of the interconnection topology with other resources), and a description of any messages sent and received by the hardware which are specific to that hardware and not due to the software running on that hardware (i.e., messages sent or received by the software which runs on the hardware are described in the software architecture). Additional fields are also provided for each design factor which is chosen to describe the resource. This allows required and budgeted values of interest to be captured. Figure 5. shows a candidate hardware architecture of the passive sonar system.

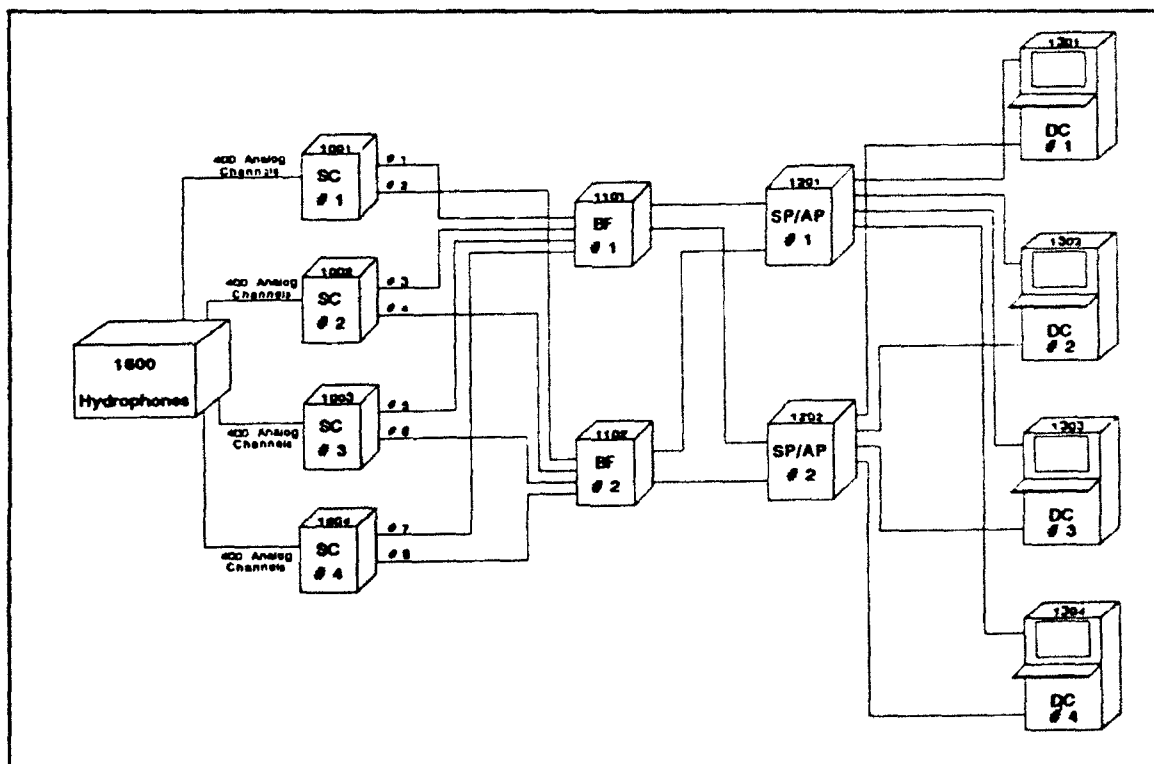


Figure 5. Passive Sonar System Hardware Architecture

Similar to hardware architecture, the software architecture includes a description of the various software modules of the system. Each module is described in terms of the processing and algorithms implemented, selected design factors addressing throughput requirements, memory requirements, etc., and a description of the messages sent and received by the module. The software architecture description is also intended to contain other information typically included in a Software Requirements Specification (SRS) as defined by the DOD-STD-2167A DID #DIMCCR 80025A. The software architecture can be represented using various graphical forms including a listing of source code modules with calling relationships, and message flow between modules, etc. The software architecture database contains the necessary information to construct these system software representations and to support modeling of system performance. Figure 6a. shows an example format of the software architecture of the detection function and figure 6b. shows a description of a software task within that architecture.

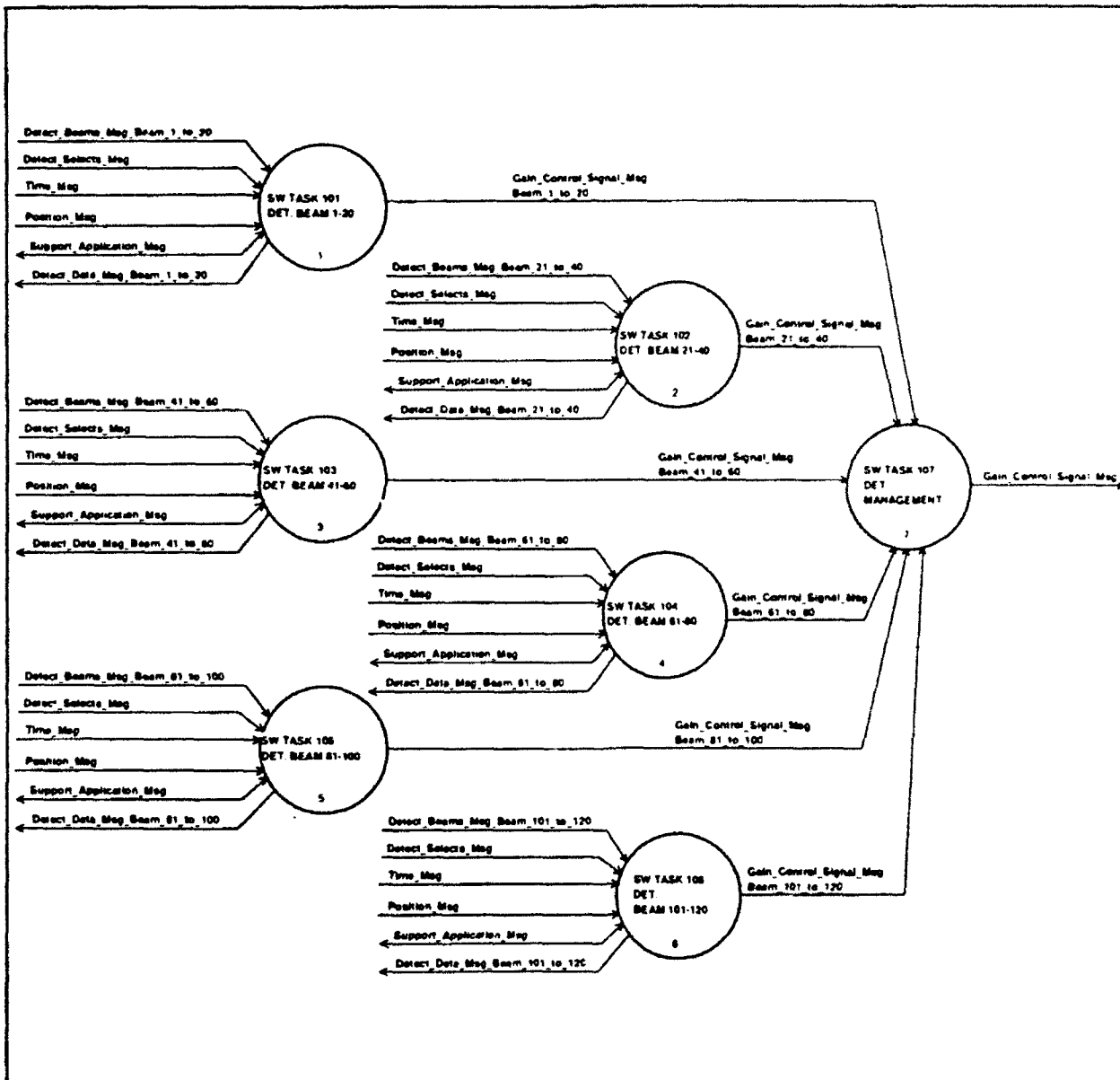


Figure 6a. Software Architecture of the Detection Function

NAME/TITLE:	SW Task 161 / Detection C&D
SUBTASK:	None
INPUT MESSAGES:	Detect_Data_Mag Sound_Mag_Op_1 Sound_Mag_Op_4 Support_Application_Mag
OUTPUT MESSAGES:	Detect_Selects_Mag Audio_Selects_Mag_Op_1 Audio_Selects_Mag_Op_2 Analyse_Control_Mag Support_Application_Mag
TASK DESCRIPTION:	This task provides the mechanism for detection generated by the system to be formatted for evaluation and analyzed.
DESIGN FACTORS:	Performance Processing Required: 0.42 MIPS Memory Required: 160 KBytes Priority: 6 Response Time: 100 msec

Figure 6b. Description of the Detection Software Task

Finally, the humware architecture describes the number and type of personnel (i.e., training, and experience levels) required to man the system under various operating conditions. In the case when human function is a large part of the system under design, the organization chart is considered as part of this architecture. Organization breakdown and information interchanges between different departments are captured with a similar format that was described in the capture of hardware or software. Figure 7. illustrates a human resource architecture of the passive sonar system and an example description of one particular operator, detection operator.

Conclusion

The above discussion is an attempt to represent the implementation aspect of the system and its relationship with the functional aspect. The issue here is not the preference of one notation over another, rather, it is the need for a robust technique to describe system resources from a detailed specification of a simple component to a high level abstraction of a complex part. With the flexibility to maintain multiple design options and the ability to analyze these options, systems engineers will have more confidence with their design. There is no doubt that the above techniques should be automated; however, early commitment in certain CASE tools or technologies can restrict the intention of the methods. With the intention to support various types of analysis, the implementation representation will also face the problem of how to automatically transform its capture information to different models of analysis.

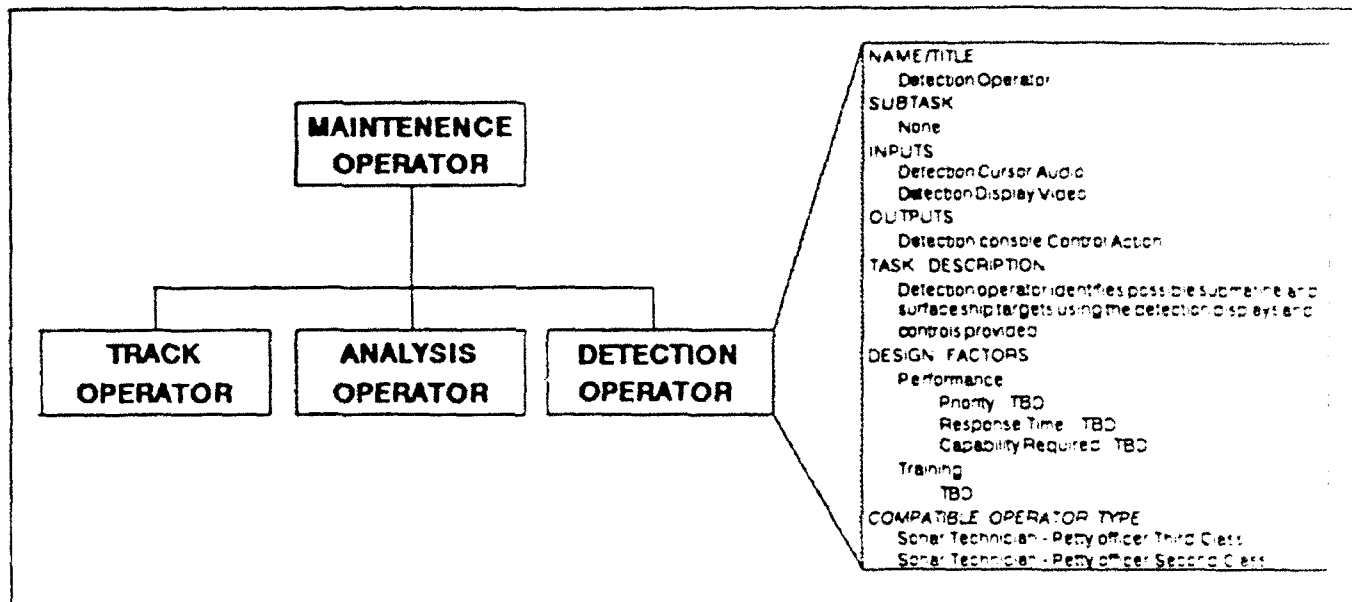


Figure 7. Human Resource Architecture of the Passive Sonar System

Acknowledgements

The authors would like to thank their sponsor, the Office of Naval Technology, especially Elizabeth Wald and Cmdr. Gracie Thompson. The authors would also like to thank Steve Howell, Michael Jenkins, and all members of the System Synthesis Specification Working Group for their contribution in the refinement of this method; and Adrien Meskin for her editorial support.

REFERENCES

- [Hoa] The Essential Views of System Development, N. Hoang, Proc. 1991 System Design Synthesis Technology Workshop, Silver Spring MD, Sept. 1991.
- [Rum] Rumbaugh, James, et al., Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, NJ, 1991.
- [You] Yourdon, Edwards, Modern Structured Analysis, Yourdon Press, Englewood Cliffs, NJ, 1989.

The Environmental Capture View: Addressing External Factors in Capture and Analysis of Large Scale Complex System Design

Nicholas E. Karangelen

Trident Systems Incorporated
Fairfax, Virginia 22030

ABSTRACT

The capture of large complex real-time system designs requires organization and representation of a large diverse body of technical information and data. While most system design capture or specification techniques address the external interfaces to the system under design, no formal, structured approach for capturing the full spectrum of external and environmental factors which impact the system design has been established. This paper addresses the preliminary definition of an Environmental Capture View within the context of a multi-domain design capture and analysis methodology. The elements of the Environmental Capture View are intended to provide a structured representation and organization of the following types of information: operational scenarios, system concept of operations, environmental conditions, external systems and interfaces, test strategies, maintenance and logistics considerations, and other external factors which impact the system under design. Candidate methods for representing and organizing selected elements are discussed with examples along with suggestions for the direction of future work in this area.

INTRODUCTION

The capture of large complex real-time system designs requires organization and representation of a large diverse body of technical information and data. Existing systems engineering tools and methodologies [1], [2], [3] offer multi-domain approaches to representing key aspects of the system design such as systems functions and data, hardware and software architecture. While most system design capture or specification techniques address the external interfaces to the system under design, no formal, structured approach for capturing the full spectrum of external and environmental factors which impact the system design has been established.

This paper addresses the preliminary definition of an Environmental Capture View within the context of a multi-domain design capture and analysis methodology as described by N.D. Hoang [4]. The five capture views defined within the methodology are briefly summarized below to provide a context for the main subject of this paper which is definition of the Environmental Capture View. The essential elements of the Environmental Capture View are defined herein, and several candidate methods for representing and organizing the information for selected elements of the Environmental Capture View is proposed.

FIVE VIEWS OF A COMPLEX SYSTEM

The central element of the multi-domain design capture and analysis methodology is definition of the multiple design domains or Capture Views of the system which address the principal system design perspectives: (1) Environmental, (2) Informational, (3) Functional, (4) Behavioral, and (5)

Implementation. The five views partition the system design into logical segments corresponding to key perspectives of the system design. The partitioning of the system design capture into these five Capture Views has evolved beginning with the NAVSWC ASW Methods and Tools Group [5] and has been described in its present form by N. D. Hoang [4]. The five Capture Views are briefly summarized in Table 1.

Table 1
FIVE VIEWS OF A COMPLEX SYSTEM

DESIGN VIEW	VIEW OBJECTIVES	DESIGN ELEMENTS
Environmental View	<ul style="list-style-type: none"> • Establish Conditions and Events Constraining System Operations • Specify Performance MOEs and Conditions of Measurement 	<ul style="list-style-type: none"> • Environmental Conditions and Event Descriptions • External System Descriptions • System Initial Conditions • Measures of Effectiveness
Informational View	<ul style="list-style-type: none"> • Characterize System Concept of Operations • Represent System Component's in Abstract Terms 	<ul style="list-style-type: none"> • Entity - Relationship Diagrams • Attribute/Method Descriptions
Functional View	<ul style="list-style-type: none"> • Define System Functions and Decompositions • Specify Data Flow Requirements 	<ul style="list-style-type: none"> • Function/Data Flow Diagrams • Process Specifications • Data Dictionary
Behavioral View	<ul style="list-style-type: none"> • Define System States and Triggers • Specify System Behavior Characteristics 	<ul style="list-style-type: none"> • Control Flow Diagrams • State Transition Diagrams • Control Specifications
Implementation View	<ul style="list-style-type: none"> • Define the Physical Hardware, Software and Human Resources Which Make up the System • Specify System Physical Interconnectivity 	<ul style="list-style-type: none"> • Hardware, Software and Human Resource Descriptions • Performance Parameters and Resource Characteristics • Function-Resource Mapping

An attempt to address all of issues associated these views simultaneously without a structured methodology is a multi-dimensional problem of a magnitude which exceeds the capacity of most if not all systems engineers. Each of these views provide key information concerning particular aspects of the system under design. Taken individually the views allow the systems engineer to partition the design and analysis of a proposed or existing system into manageable parts.

The capture approach for these design domains or views share a common hierarchial structure which supports management of the magnitude and complexity associated with a large system design. Flat representations of complex system designs rapidly become unwieldy as the design detail unfolds. A hierarchial structure allows the system views to be represented at various levels of detail from a broad top level which encompass the breadth and scope of the system and its external interfaces to very low levels which describe the details of a particular segment of the system design.

Each design view represents the system from a particular perspective and highlights different aspects of the design, however they are not independent. The views represent different aspects of the same system and therefore must be consistent. The features of a particular view can directly or indirectly impact, to a greater or lesser degree, the design in another view depending on how the relationship between the views is specified.

SYSTEM DESIGN CAPTURE AND ANALYSIS AUTOMATION

Successful employment of a multi-domain design capture and analysis methodology in supporting a complex system design is largely a function of the degree of mechanization which can be achieved. The size and complexity of large scale advanced computer systems render manual application of any design process or method unusable. Considerable potential benefits can be gained from automation which supports a disciplined structured capture of the initial iteration of a system design and subsequent editing of that capture. Further significant efficiencies can be gained through automated consistency and completeness checking within and between the five system views which represent the captured design. However, in light of the overwhelming systems engineering task represented by analysis of an advanced complex system design, the most significant productivity gains are in automated support for design simulation and analysis within an integrated and highly automated design capture and analysis environment.

DEFINING THE ENVIRONMENTAL CAPTURE VIEW

The Environmental Capture View is defined as the structured representation and organization of the following types of information: operational scenarios, concept of operations, environmental conditions, external systems and interfaces, test strategies, maintenance and logistics considerations, and other external factors which impact the system under design. The information captured in the Environmental Capture View is necessary to address important issues of a system under design but may not typically be included in the design itself. The key elements of the Environmental Capture View are listed below with a brief description. The following sections address each of these elements in some detail including selected examples.

Operational Scenarios describe situations and sequences of external events which the system must address. The most stressing cases and most likely cases are identified and described from the potentially large spectrum of possible operational scenarios.

The Concept of Operations describes the proposed approach for operation of the system under design from the operators perspective.

External Systems and Interfaces to the system are captured including information necessary to develop an external system model to support system design simulation and testing.

The Environmental Conditions under which the system must operate may include geographic, meteorologic, electromagnetic, and acoustic environmental factors as well as others associated with the systems operational surroundings.

System Test Strategies for testing system compliance with the top level system requirements are captured including system performance metrics, system test approach and test procedures, system Sim/Stim, and test instrumentation.

Maintenance and Logistics considerations associated with the system through its life cycle which impact the design are also captured.

Other External Factors which influence and affect the system under design are also documented such as design constraints and guidance imposed by the program development sponsor.

The elements of the Environmental Capture View including the external interfaces and environmental conditions are described and captured in terms which are compatible with the Informational, Functional, Behavioral, and Implementation Capture Views such that the consistency across the various capture views can be analyzed. These descriptions also establish a basis for development and analysis of a performance simulation for the system under design. Simulation provides the ability to identify potential shortfalls and errors in the system design early in the design cycle when changes and corrections are considerably less costly. The descriptions also provide important information needed for system integration and testing as well as for development of system external interface specifications.

It is important to note that this paper does not address a particular systems engineering design process. Instead, it describes a technique and methodology for capturing the Environmental Capture View of a complex system design regardless of the systems engineering process model employed. This paper describes the current evolution of the Environmental Capture View and represents a snapshot of an ongoing effort to formalize the elements of the Environmental Capture View and the techniques for documenting those elements. Examples extracted from a passive sonar system sample problem [6] are employed in this paper to illustrate the capture techniques and to describe the rationale for the methods employed.

OPERATIONAL SCENARIOS

This element of the Environmental Capture View identifies and describes the operational scenarios which are expected to be encountered by the system under design both in the near-term and far-term over the system's projected life. The capture of operational scenarios is intended to establish bounds on the spectrum of possible operational scenarios, identify key scenarios which represent the most likely and most stressing cases for the system under design, and describe selected key scenarios in detail. These key scenarios are captured in sufficient detail to establish test cases for system performance simulation and analysis.

Scenarios captured using the approach described in this section could be maintained in a scenario library and reused for multiple system designs where applicable. An example of where this approach would be particularly useful is the set of seven scenarios promulgated recently by the U.S. military Joint Chiefs of Staff (JCS). This set of seven scenarios has been approved for use in establishing system development guidance and provides a vehicle for evaluating the warfighting utility of existing and proposed systems.

SPECTRUM OF OPERATIONAL SCENARIOS

The first component to be addressed in capturing the operational scenarios is a mechanism for bounding and describing the wide spectrum of possible operational scenarios for the system under design. The set of possible scenarios is, in general, very large and can not be efficiently represented by a list of scenario cases. The approach we will employ here is to establish the key parameters which characterize the scenarios and then to establish the range of possible cases for each parameter. These parameters provide a mechanism to characterize entire classes of scenarios and to identify the most likely and most stressing cases to support system simulation and analysis.

Operational scenarios can be characterized in many ways such as the geographic setting, environmental conditions, the objectives of the system users, the intensity of operations, etc. A set of parameters must first be identified and defined which describe the essential elements of an operational scenario for the system under design. These parameters should be selected to be orthogonal if possible to allow any combination of the various parameters to be selected. A range of values or cases for each scenario parameter is also identified. The goal is to establish a set of scenario parameters and parameter values which can uniquely characterize any possible scenario which the system under design may encounter.

A simplified set of scenario parameters for the passive sonar system sample problem is illustrated below. Five parameters have been identified to describe the spectrum of operational scenarios which the sample passive sonar system is intended to operate and which can be used to uniquely identify a particular scenario:

(1) Tempo of Operations - Defined as the level of operational intensity associated with the platform mission and the system's role in supporting that mission. The range of possible cases for the Tempo of Operations scenario parameter is illustrated as follows:

- Low - Independent transit in international waters
- Moderate - CVBF screening operations in open ocean
- High - Threat submarine tracking operations

(2) Level of Conflict - Defined as the combat readiness or alert state of the platform and related to the likelihood of hostile actions. The range of possible cases for the Level of Conflict scenario parameter is illustrated as follows:

- Peacetime
- Crisis Response
- Transition to War
- Regional Conflict
- Global Conventional War

(3) Environmental Acoustic Conditions - Defined as the description of prevailing environmental conditions which affect acoustic sensor performance. A simplified range of possible cases for the Environmental Acoustic Conditions scenario parameter is illustrated as follows:

- High Ambient Noise/High Propagation Loss
- High Ambient Noise/Moderate Propagation Loss
- High Ambient Noise/Low Propagation Loss

Moderate Ambient Noise/High Propagation Loss
 Moderate Ambient Noise/Moderate Propagation Loss
 Moderate Ambient Noise/Low Propagation Loss
 Low Ambient Noise/High Propagation Loss
 Low Ambient Noise/Moderate Propagation Loss
 Low Ambient Noise/Low Propagation Loss

(4) Acoustic Contact Density - Defined as the number of contacts within acoustic detection range of the system. A simplified range of possible cases for the Acoustic Contact Density scenario parameter is illustrated as follows:

Low - less than 3 simultaneous acoustic contacts
 Moderate - 3 to 10 simultaneous acoustic contacts
 High - 10 to 15 simultaneous acoustic contacts
 Very High - greater than 15 simultaneous acoustic contacts

(5) System Operating Mode - Defined as the current readiness state of the system. A simplified range of possible cases for the System Operating Mode scenario parameter is illustrated as follows:

Full-up - All functional capabilities available
 Degraded Mode - Only detection and limited tracking capabilities available

The scenario parameters and the ranges of possible cases or values for each scenario parameter are then configured in a scenario matrix which allows the user to summarize the spectrum of possible scenarios in a compact form. Individual scenarios can easily be extracted from the scenario matrix by selecting one value or case from each of the scenario parameters in the scenario matrix. This also creates a short hand method for labeling scenarios.

Tempo of Operations	Low	Medium	High						
Level of Conflict	Peacetime	Crisis Response	Transition to War	Regional Conflict	Global Conventional War				
Environmental Acoustic Conditions	High AN High PL	High AN Mod PL	High AN Low PL	Mod AN High PL	Mod AN Mod PL	Mod AN Low PL	Low AN High PL	Low AN Mod PL	Low AN Low PL
Acoustic Contact Density	Low	Moderate	High	Very High					
System Operating Mode	Full-up	Degraded Mode							

Figure 1 Passive Sonar System Example Scenario Matrix

The scenario matrix for the passive sonar system is illustrated in Figure 1. Even in this simplified example the number of possible scenarios described by this scenario matrix for the passive sonar system totals 1080 (3x5x9x4x2). Describing in detail 1080 scenarios is not a reasonable approach to take nor is it necessary. The next two sections describes a method for selecting a small subset of the possible scenarios which represent the most likely scenarios and the most stressing scenarios. These few selected scenarios will then be described in detail and will form the basis for system analysis and testing.

MOST LIKELY SCENARIOS

The set of most likely (or most common) scenarios which the system under design will encounter is derived by examining the spectrum of possible scenarios as embodied in the scenario matrix. The most likely range for each of the scenario factors is determined and then the most likely combinations of the parameters are identified. Understanding the set of most likely scenarios represents an important aspect in developing the system design. These scenarios also define one set of test cases which can be used to simulate and analyze the system under design.

Tempo of Operations	Low	Medium		High						
Level of Conflict	Peacetime	Crisis Response		Transition to War		Regional Conflict		Global Conventional War		
Env Acoustic Conditions	High AN High PL	High AN Mod PL	High AN Low PL	Mod AN High PL	Mod AN Mod PL	Mod AN Low PL	Low AN High PL	Low AN Mod PL	Low AN Low PL	
Acoustic Contact Density	Low	Moderate		High	Very High					
System Operating Mode	Full-up		Degraded Mode							

Figure 2 Scenario Matrix With Most Likely Cases Highlighted

Based upon combining the most likely scenario parameters identified in Figure 2, the following two scenarios represent the most likely scenarios for the sample sonar system:

Likely Scenario #1 - A moderate tempo of operations in a peacetime situation in a moderate acoustic environment with moderate contact density and the system in the full-up operating mode.

Likely Scenario #2 - A moderate tempo of operations in a crisis response situation in a moderate

acoustic environment with moderate contact density and the system in the full-up operating mode.

MOST STRESSING SCENARIOS

The set of most stressing scenarios which the sample passive sonar system will encounter is also derived by examining the spectrum of possible scenarios. The most stressing cases for each of the scenario factors is determined and then the most stressing combinations of the five parameters are identified. Understanding the set of most stressing scenarios represents an important aspect in developing a system design which is operable and meets the intent of the system requirements. These scenarios also define a second set of test cases which can be used to simulate and analyze the system under design.

Tempo of Operations	Low	Medium	High						
Level of Conflict	Peacetime	Crisis Response	Transition to War	Regional Conflict	Global Conventional War				
Env	High AN	High AN	High AN	Mod AN	Mod AN	Mod AN	Low AN	Low AN	Low AN
Acoustic Conditions	High PL	Mod PL	Low PL	High PL	Mod PL	Low PL	High PL	Mod PL	Low PL
Acoustic Contact Density	Low	Moderate	High	Very High					
System Operating Mode	Full-up	Degraded Mode							

Figure 3 Scenario Matrix With Most Stressing Cases Highlighted

Based upon combining the most stressing scenario parameters identified in Figure 3, the following scenarios represent the most stressing scenarios for the sample sonar system:

Stresing Scenario #1 - A high tempo of operations in a regional conflict situation in a high AN/PL acoustic environment with very high contact density and the system in the full-up operating mode.

Stresing Scenario #2 - A high tempo of operations in a transition to war situation in a high AN/PL acoustic environment with very high contact density and the system in the full-up operating mode.

Stresing Scenario #3 - A high tempo of operations in a regional conflict situation in a high AN/PL acoustic environment with very high contact density and the system in the degraded operating mode.

Stresing Scenario #4 - A high tempo of operations in a transition to war situation in a high AN/PL acoustic environment with very high contact density and the system in the degraded operating mode.

SCENARIO DESCRIPTIONS

The system scenario descriptions are captured in a graphic and textual format which is tailored to support documentation of the information and data required to fully describe the conditions, events and other features of a given scenario. The following outline provided in Figure 4 represents a preliminary baseline format for capturing scenarios associated with large scale military systems. This baseline format is intended to be tailored for use in specific programs. An example scenario description is currently being developed using this approach for the sample sonar system described in reference 5.

- 1.0 Introduction
- 2.0 Scenario Characterization and Definition of Terms
- 3.0 Mission Description Summary/Mission Success Criteria
- 4.0 Geographic Area of Interest/Assumptions
- 5.0 Own Force Organization, Platforms, Objectives and Operations
- 6.0 Threat Organization, Platforms, Objectives and Operations
- 7.0 Environmental Conditions
- 8.0 Initial Conditions and Chronology of Key Events
- 9.0 Glossary
- 10.0 References

Figure 4 Scenario Description Contents

CONCEPT OF OPERATIONS

The system concept of operations describes a high-level philosophy for the conduct of specific operations employing the system under design. During the early phases of the system development process, the concept of operations description is developed such that it applies to a broad spectrum of possible system implementations and is not constrained by a specific candidate system design, personnel manning plan, hardware/software implementation or existing system operator-machine interfaces. The focus is on describing the concept for mission execution using the system at a high level without regard to a specific physical system implementation. As the system design is developed over time, the level of detail captured in the concept of operations can then be refined to address implementation specific design features.

Complex systems, and in particular military systems, often are designed to address a wide spectrum of operational scenarios and therefor may have a correspondingly robust concept of operations. The concept of operations for a complex system may have several variations which are

a function of the particular type of scenario which is being addressed. The system concept of operations as captured in the Environmental Capture View is not intended to address the entire spectrum of possible operational scenarios but will be described for a particular scenario or class of scenarios. In general, one basic concept of operations associated with a selected scenario (e.g. one of the most likely or most stressing cases) will be captured with variations to describe the principal differences in the operations for the key scenarios under consideration (e.g., the most-likely and most-stressing scenario cases).

The technique currently identified for capturing the system concept of operations is a structured english text document augmented by time lines which capture control and display utilization and operational sequence diagrams. The outline for the concept of operations document is provided in Figure 5. As with the scenario description document, the format is intended to be tailored for use in specific programs.

- 1.0 Operator Machine Interface Configuration Description
- 2.0 System Modes of Operation
- 3.0 Operator Relationships and Activities
- 4.0 Control and Display Utilization
- 5.0 Operational Sequence Diagrams

Figure 5 Concept of Operations Document Outline

Additional work is required in this area to identify alternative formal methods for capturing system concepts of operations which may be more compatible with: (1) future design capture automation techniques; (2) graphical system representation and display methods; and (3) requirements and design rational traceability techniques.

EXTERNAL SYSTEMS AND INTERFACES

This element of the Environmental Capture View identifies and describes the external systems which have some relationship to the system under design as well as the external interfaces from these external systems to the system under design. The external interfaces between the system under design and the natural environment (e.g. temperature and pressure transducers) and are also described. The principal purpose in describing the external systems is to provide a means for simulating their behavior and modeling their interfaces to the system under design. This information is necessary to create an external model which can be used for the purpose of modeling performance of candidate design implementations during early design phases or for interface stimulation during testing of the system.

This element of the Environmental Capture View also serves to summarize the system boundaries, external interfaces to the system, and describes how the system under design fits into the larger architecture and organization of the higher entity of which the system is a part. It does not address the internal structure of the system, but serves to define the system's interfaces and relationships to

other systems and activities which are considered outside the scope of the system under design. It also describes and defines the objects external to the system and the behavior of those objects in relationship to the system under design.

Information captured concerning those systems which are external to the system under design is driven by a spectrum of activities from providing a means for simulating external system behavior in support of modeling the performance of candidate system designs, through simulating and stimulating external interfaces during testing of the actual system during integration. The complexity of an external model developed as a "simulation harness" is driven by the level of fidelity required to stimulate the internal model of the system under design. The external model should support the system design process from an early top level representation to a very detailed level of design. The capture techniques employed to represent the external systems must therefore be flexible and support evolution from an abstract level to a very detailed level of fidelity. The capture approach should also be compatible with the approach employed to capture the system under design to ensure that internal and external simulation models created from design capture activities are easily integrated.

Based upon the forgoing considerations, an obvious candidate approach for capturing external systems is to employ the same techniques used to capture the system under design summarized earlier in this paper. While this would provide the extensibility and compatibility features needed, it would also be costly to create the five capture views for each external system. This reasoning has led to tentative selection of the Implementation Capture View as the basis for capture of external systems. It is important to note that the external interfaces of the system under design (which are considered part of the system under design) are intended to be captured using the full five capture views. Further investigation is required to determine if an adequate external model representing the external systems can be constructed from the information contained in the Implementation Capture View.

The current method for documenting the Implementation Capture View is described by N.D. Hoang [7]. This approach for capturing external systems would potentially provide the benefits of using a common approach for design capture and modeling at an acceptable cost. Some extensions and modifications to the Implementation Capture View as currently envisioned may be required and will be identified as this work continues. An example capture of external systems using this technique is currently being developed using this approach for the sample sonar system described by Karangelen and Hoang [6].

ENVIRONMENTAL CONDITIONS, SYSTEM TEST STRATEGY, AND SYSTEM MAINTENANCE AND LOGISTICS CONSIDERATIONS

These three elements of the Environmental Capture View include additional key information which can have considerable impact on the system under design. The environmental conditions represented by prevailing meteorological, electromagnetic, and acoustic conditions is often a key factor in the performance of sensors such as radar and sonar, and in the performance of communications systems. The system test strategy and system maintenance and logistics considerations can typically impose design constraints which must be identified early in the design process to avoid the potentially high cost of back-fitting test and maintenance capabilities in the late stages of system integration and testing. To date no formal method for the capture of these elements of the Environmental Capture View have been established. Existing techniques for development and capture of system test strategy and system maintenance and logistics

considerations are currently being reviewed. No formal capture techniques for environmental conditions in general have been identified to date however, considerable data is available from a variety of sources for specific classes of systems such as active and passive acoustic underwater sensors, as well as land and air based radar. One potential method for organizing and capturing the environmental data is based on categorizing the various environmentally dependent technologies and establishing a format for each specific technology area (e.g. radar, sonar, etc.)

FUTURE WORK

This paper describes a preliminary description of an Environmental Capture View within the context of a multi-domain design capture and analysis methodology. Further refinement of the techniques for characterization and selection of operational scenarios, description of system concept of operations, and capture of external systems and interfaces is required to: (1) address the employment of automated design capture and simulation methods, (2) enhance compatibility of the Environmental Capture View with other design views, and (3) to provide a mechanism for traceability of system requirements and design rational. Capture techniques have yet to be specified for environmental conditions, system test strategy, and system maintenance and logistics elements of the Environmental Capture View.

ACKNOWLEDGEMENTS

This paper contains many of the concepts initiated by the ASW Methodology and Tools group under the direction of Janis Bilmanis of the Naval Surface Weapon Center (NAVSWC) White Oak, Maryland. This work also reflects concepts developed and refined in cooperation with the System Synthesis Specification Working Group (SSSWG) at NAVSWC. The author would like to recognize the contributions of Ngocdung T. Hoang, Mike Edwards, Steven Howel, and David Britton. The Space and Naval Warfare Systems Command (SPAWAR), NAVSWC, the Office of Naval Technology (ONT), and Trident Systems Incorporated have provided support for this and related research by the author.

REFERENCES

1. Technical Documentation Department of Cadre Technologies Inc. *teamwork/SA User's Guide, Release 4.0*. Cadre Technologies Inc., Providence, RI, Dec 1990.
2. Tom Demarco. *Structured Analysis and System Specification*. Prentice-Hall, Inc., Yourdon Press, Englewood Cliffs, NJ, 1979.
3. Derek J. Hatley and Imtiaz A. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House Publishing, New York, NY, 1987.
4. N. Hoang. *The Essential Views of the System*. 1991 Systems Design Synthesis Technology Workshop, Silver Spring, MD, September, 1991.

5. ASW Methodology and Tools Working Group. *ASW Architecture Development and Analysis Methodology, Version 1.0*. Technical Report, Honeywell Marine Systems Division, Everett, WA, March 1990.
6. N. Karangelen and N. Hoang. *Sample Sonar System Example Problem Capture*. 1992 SSSWG Meeting Review Copy, Silver Spring, MD, February 1992.
7. N. Hoang and N. Karangelen. *A View to an Implementation*. 1992 Systems Design Synthesis Technology Workshop, Silver Spring, MD, July 1992.

A STUDY FOR THE DEVELOPMENT OF A REQUIREMENTS TRACEABILITY MODEL

Balasubramaniam Ramesh

Michael Edwards

Code AS/RA
Naval Postgraduate School
Monterey, CA 93943
E-mail: ramesh@isl.as.nps.navy.mil

Code U33
NSWCDD
Silver Spring, MD 20903-5000
E-mail: medward@nswc-wo.nswc.navy.mil

1 Introduction

A primary concern in the development of large-scale, real-time, complex, computer-intensive systems is ensuring that the performance of the system meets the specified requirements. As part of the system development and maintenance process, many decisions and trade-offs are made that affect a variety of components of the system. Further, the requirements themselves evolve and undergo many changes during the development process. In such a context, it is essential to maintain traceability of requirements to various outputs or artifacts produced during the systems design process, to ensure that the system meets the current set of requirements. Maintaining consistency between the requirements and the design is especially critical in situations where an organization relies upon outside contractors for developing systems. Having a systematic way of validating that every requirement is met by the design is important, not only to ensure that the system performs correctly, but also to determine whether contractual obligations have been met.

It should be noted that throughout this paper, the term *design* is used to refer to any activity that leads to the creation of artifacts. Potts and Bruns [1] note that even the early phases of the systems development process involve the creation of intermediate artifacts and the term *design* could be used to denote such activities as well.

A comprehensive scheme for maintaining traceability, especially for complex, real-time systems, requires that *all* system components (not just software), created at various stages of the development process, be linked to the requirements. These components include hardware, software, humanware, manuals, policies and procedures. In order to achieve this objective, it is essential that traceability be maintained through various phases of the systems development process, from the requirements as stated (or contracted) by the customer, through analysis, design, implementation and testing to the final product.

In the next section, we discuss past research and current tools for traceability. Next, we discuss our approach towards developing a model of traceability. We describe an initial empirical study and preliminary results that are being used in the design of future work. Finally, we present an example of a complex traceability relationship based on a model (design) rationale in requirements engineering and address some of the issues raised in our study.

2 Background

2.1 Definition of Traceability

A variety of definitions of traceability have been proposed in the literature depending on the intended use of traceability information. Greenspan and McGowan [2] provide a generic definition of traceability: traceability as a property of a system description technique that allows changes in one of the three system descriptions- requirements, design specifications, implementation - to be traced to the corresponding portions of the other descriptions. The correspondence should be maintained thought the lifetime of the system.

Schneidewind defines traceability as the ability to identify the technical information which pertains to a software error which has been detected during the maintenance phase and thereby trace the error to the applicable design specifications and user requirements [3], [4].

The need to provide traceability is recognized in most critical standards governing the development of software for the U.S. Government. For instance the DoD-STD-2167A specifies that "the contractor shall document the traceability of the requirements allocated from the system specification to each Computer Software Configuration Item (CSCI), its Computer Software Units (CSUs), and from the CSU level to the Software Requirements Specifications (SRSs) and Interface Requirements Specifications [5],[6]". An elaboration of this requirement states that "the Software Design Document describes the allocation of requirements from a CSCI to its CSCs and CSUs[6].

It should be noted that even this elaboration is not specific about the nature of the linkages to be maintained and leaves the interpretation of the meanings of such linkages to the users. Unless the semantics of the linkages are clearly specified, the existence of a link between a CSCI to its CSCs could denote one of several possibilities including : the requirements have been completely allocated, some of the CSCs satisfy some aspects of the requirements, it is possible to verify that the requirements have been completely satisfied.

Many of the above definitions are geared toward maintaining traceability in software components. Our goal is to develop a model that will address traceability issues at the level of systems design, relating requirements to all system components. Further, such a model should not only discuss the kinds of linkages or relationships that should be maintained, but also the reasoning that can be performed with such traceability information.

2.2 Why Traceability?

As discussed earlier, requirements traceability is imperative to ensure the closure of all systems components [7].

As requirements tend to evolve over the long life of large scale systems, maintaining systems to meet such evolving needs is critical. As large scale systems are composed of interdependent components and as the design process "spreads information", even small changes at the level of requirements may lead to major changes to various parts of the system. Requirements traceability will go a long way in alleviating the problem of maintenance by facilitating the identification of interdependencies among components and localizing the effects of changes made at various levels of systems design. Further, if the relationships between design and the requirements can be maintained, any change to the design can be

analyzed to determine if the system still meets every requirement. Since every requirement that is affected by a part of the design can be identified, the side effects of changes can be contained and avoided.

Arguing for the importance of traceability, Choi and Scacchi state that the correctness of a configured software description (i.e., a software system's life cycle descriptions) can be formalized in terms of their consistency, completeness and traceability [8].

Traceability is required irrespective of the software design methodology or the hardware software architectures used in a project. Some methodologies provide tight relationships between components produced during various stages of the design process and hence automatically provide traceability. An extreme example is the development of software based on formal specifications. The formal requirements are transformed into executable systems. The transformation history provides traceability between formal requirements and the executable system. Automated code generation using a fourth generation language is an example of such 'automatic' traceability which is limited in scope.

A recent workshop on reuse in practice concluded that an environment to facilitate reuse must support automatic traceability of a component through the requirements to the executable components. Traceability is important for user understanding of the component's design and implementation, since it captures the context and the constraints of the development process and that this understanding assists the user of a component in reusing it in another situation [9].

3 Traceability Tools

Initial work on traceability concentrated on providing document traceability. Document traceability determines the existence of relationships between two document components [10], [11].

ARTS [12] is among the earliest systems to capture and use of traceability information. The current commercial state-of-the-art requirements traceability tools (i.e., employed in tools such as Teamwork/RQT, R-trace, RDD-100) simply link requirements to pieces of the design and implementation. Current tools, similar to ARTS, tend to focus on the database management issues related to maintaining links between requirements and various components of the system. An area that is not adequately addressed by current approaches is the capture and use of the semantics of the relationships. For instance, these techniques do not address the issue of representing how the requirement is satisfied by the design, but just facilitate capturing the fact that some relationship exists.

Another shortfall with today's traceability tools is that they lack the ability to trace back from the actual pieces of design and implementation to the requirements. Although some tools such as Teamwork/RQT and R-trace allow the user to trace from requirements analysis tools such as Teamwork and Software Through Pictures, they do not have a method for tracing from a particular piece of hardware or humanware back to the requirements. This capability would be extremely useful in performing systems maintenance.

4 Towards a Model of Traceability

The need for better understanding of traceability is widely recognized. As discussed in the previous section, maintenance of traceability is also mandated by several standards. However, the precise definition of the kinds of traceability linkages or relationships that must be maintained is currently lacking. A major challenge in this research is the development of a model that represents and provides the semantics of various traceability linkages or relationships between requirements and system components.

There are a variety of stakeholders involved in the systems development process, including project sponsors, project managers, analysts, designers, maintenance personnel, testing personnel, and end users. A basic premise in our research is that the development of a model of traceability could be geared towards the needs of these various stakeholders at various stages in the systems development process. The first phase of our approach to this problem has been an empirical one. We have conducted an initial empirical study to explore the traceability needs of various stakeholders. The results of this study are being used in designing a comprehensive study involving stakeholders in large scale, complex, real-time systems development efforts. In this paper, we present the details of the initial study and some preliminary findings which will be explored in the follow-up studies.

Study Design Our data collection strategy in the initial study involved a two-pronged approach: focus groups interviews for idea generation & evaluation and protocol analysis of problem solving behavior.

Subjects

The subjects in the study came from a Masters program in Information Technology at the Naval Postgraduate School. The study was conducted after the students had completed the analysis, design and implementation of an information system based on a case study. The case study was developed based on a real-life large scale project and had been successfully used in similar studies [13]. The case analysis involved a variety of data gathering methods during the analysis phase including informal descriptions of user needs, simulated client meetings, and actual documents from real-life situations. The major outputs developed by the participants included requirements statements, data flow diagrams, entity-relationship diagrams, database design and implementation. These activities were completed during a period of over two months prior to the subject's participation in the focus groups. Many subjects had extensive experience in domains other than computer based systems development such as ship building and aviation maintenance where concepts of traceability are widely used.

Task

The case study was in the domain of customer order processing in a utility company. The problem was chosen for several reasons:

1. The case study has been developed based on data from an extensive domain analysis. The domain analysis was based on a real life system developed by a large information systems consulting organization.
2. The case study has been used successfully in several settings including protocol analysis of group problem solving behavior.

3. The problem domain is familiar to the subjects as they have had personal experiences with the services provided by the system.
4. Real life data could be easily collected from a utility company and used in the analysis and design of the system when necessary (e.g., rate schedules were collected from the local utility company and used in systems design)
5. The problem is sufficiently complex to cover all the basic elements of systems design
6. The problem could be partitioned so that different groups of students could be assigned projects that could be completed within a reasonable time frame.

Focus Groups

Focus group interviewing is among the most frequently used form of qualitative data collection technique. Focus group interviews are widely used in several domains including market research. One of the major purposes of focus groups is idea Generation.

Setting Focus groups were conducted in a relatively formal setting - a group meeting room equipped with facilities for audio/video recording. Each focus group consisted of 5-10 panelists/students and the following steps were involved in each session:

- A short warm-up period during which everyone, including the moderator, got introduced and the ground rules of the interview stated.
- This was followed by a predisposition discussion, about the contexts in which the traceability issues needed to be explored. This included general discussions on the stakeholder interests in traceability.
- Relevant material on traceability from current vendor announcements and research briefs were presented. These provided the basis for further discussions on their strengths and weaknesses as well as modifications/extensions needed on current approaches.
- After all material had been discussed, a collective and comparative discussion of all topics was conducted. This was followed by a wrap-up of the discussion. During the wrap-up session, the participants were prodded for their summaries of what was discussed in the group meeting.

Protocol Analysis A study of the problem solving behavior of subjects engaged in a traceability exercise was conducted. The primary source of data was the verbal protocols of subjects. The verbal protocols provide a trace of the thinking process in arriving at a solution. The subjects in this exercise were required to identify traceability information that could be incorporated in their projects to satisfy various stakeholders.

Future Research

The major purpose of the above mentioned studies is to provide the basis for conducting an empirical study in real-life systems development environments. The outcomes of the current studies will help in the design of questionnaires as well as the design of focus groups and structured interviews with "real" stakeholders in systems development (designers, analyst, end users etc.)

Comparison of the two primary data collection strategies provides some interesting insights into the appropriate research methodology for future work. Focus groups provided surprisingly interesting results. In an exploratory data collection method, the researcher's biases do not constrain the participants. In our study, for instance, many participants related concepts of requirements traceability to their experiences in ship-building and aircraft maintenance which employ similar concepts. Focus groups conducted with participants who have real-life systems development experience is likely to provide very valuable sources of information. As the participants are not restricted by the researcher's ideas and predisposition, this methodology will often provide new perspectives and approaches to the problem being explored. Protocol analysis, on the other hand, is likely to provide detailed information on a problem solving task in which the participant has sufficient knowledge. However, it is extremely expensive in terms of demands on the subjects and the researcher and may involve extensive work in study design. Therefore, the use of this methodology should be restricted to a very small number of subjects.

4.1 Issues in the development of a model of Traceability

In the following section, we discuss some preliminary findings that would help develop a model of traceability and mechanisms to support capturing and reasoning with this information. These findings suggest that several areas need to be addressed by future research. Several examples drawn from prior research have been included to elaborate the major issues.

- Different stakeholders are likely to have different uses for a given traceability linkage or relationship between system components. Further they may also need different type of traceability linkages between the same systems components. For instance, the traceability linkages between a requirement and an implementation may denote that the implementation *satisfies* the requirement. The end user may be primarily interested in using this information in ascertaining that the system meets his or her requirements, whereas, the testing personnel may be interested in deriving this linkage from the linkages between requirements and tests procedures (e.g., *test*) and the relationships between implementation and test procedures that 'validate' the implementation. If the traceability information can be used to verify whether the tests were valid and comprehensive and that the tests fully validate that the implementation meets all the test criteria, then the implementation can be thought to 'satisfy' requirements.

The above perspective elaborates on traceability as a measure of quality of the system. Quality, as viewed by the customer, is the degree to which the products complies with their needs [14]. From a project manager's perspective, a major purpose of quality assurance is to ensure that "projects are proceeding on schedule, within budget and in a traceable manner, and in accordance with customer requirements and performance criteria [15]". The linkage 'satisfies' as defined above to satisfy an end-user is unlikely to be of much interest to the program manager who can not wait until the testing phase, but should ascertain whether intermediate components such as designs meet the requirements. It is obvious that the concept of quality of a system will be different from the perspectives of different stakeholders. Various ilities define quality from the

perspectives of different stakeholders; hence the need for different types of traceability information.

- The design process spreads information; i.e., several components may be necessary to satisfy a requirement. As the system evolves over its development cycle, it is desirable to identify design or implementation elements that 'partially satisfy' a given requirement. For instance, a hardware-software combination is often necessary to satisfy a given requirement. When either the hardware or software component is developed, traceability information should reflect the fact that it partially satisfies the requirement. Such information can be used in ensuring that the partially satisfied requirements are fully satisfied by performing necessary actions.
- A corollary to the above is that it should be possible to identify a combination of design elements that 'satisfy' a requirement or are 'generated by' a requirement.

An example of such a traceability scheme is the use of AND-OR graphs to represent traceability linkages. AND-OR graphs can be used to model a task in terms of a series of goals and subgoals. Figure 1 illustrates such a complex linkage.

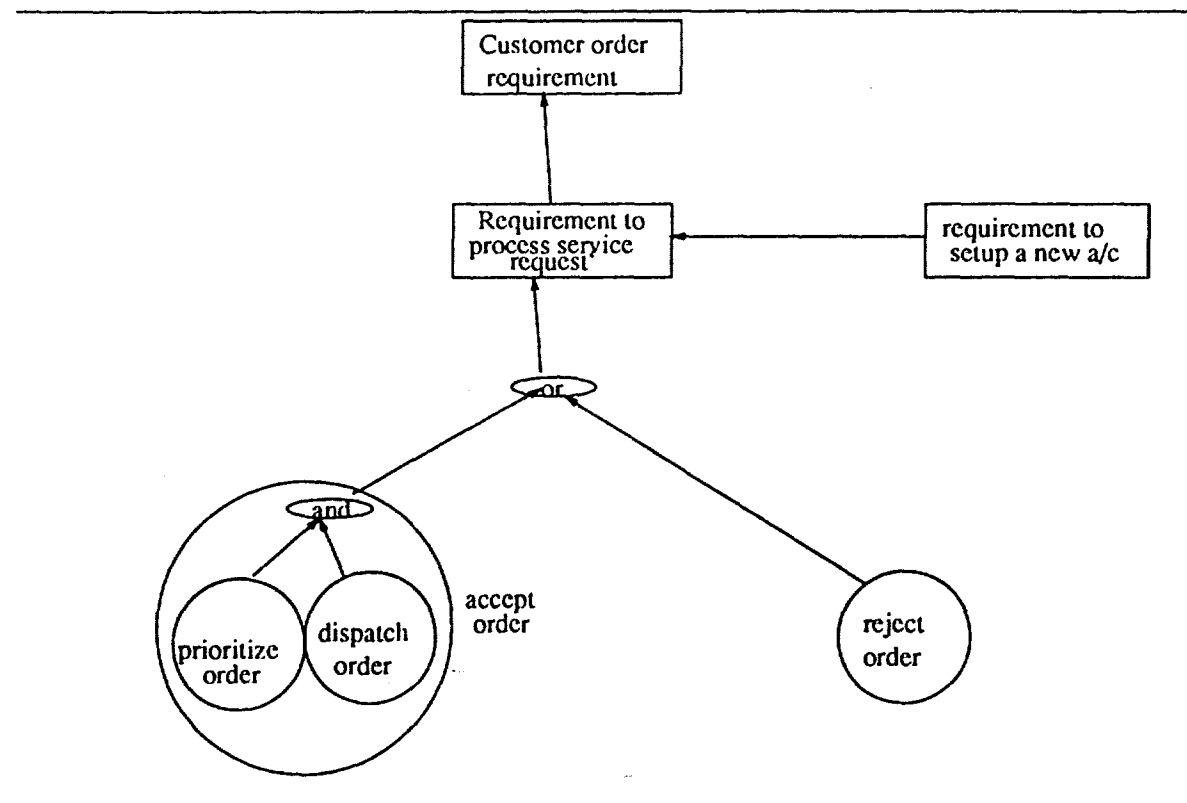


Figure 1: An example of Complex Linking

- A traceability scheme should recognize that all requirements are not equal in terms of levels of importance or significance or criticality. It may be unnecessary or even undesirable (considering the overhead involved in maintaining traceability) to maintain linkages between every requirement with every artifact created during systems design

process that is related to requirement. It is essential to identify critical requirements and maintain traceability linkages from those requirements to system components.

- A useful way of identifying the critical requirements is to relate them to the central 'mission' of the system. The business processes that *generate* requirements should be identified and requirements evaluated with respect to them to arrive at a classification. I.e., Traceability should address the issue of how the requirements are arrived at. This necessitates a mechanism to represent the *elaboration* and *refinement* of requirements from the central mission or business processes that *generate* requirements.
- A traceability scheme should allow the linkages to be qualified to denote whether the link can be *verified* formally or informally. Consider the link from a requirement to a design object that *satisfies* the requirement. In a complex, large, real-time system ascertaining whether some requirements have been satisfied can be done only qualitatively. It is often impractical or impossible to comprehensively and formally test such requirements. It is especially true of "generic" requirements an example of which is "the system shall provide a user friendly" interface. A link from such a requirement to a design or implementation component should identify not only whether the requirement is satisfied, but also "how". This information can be in the form of a link to a test procedure or specification or a qualitative evaluation by the user. From a maintenance standpoint, such *qualitatively satisfied* requirements may need periodic examination to ensure that they are still valid with changing requirements and user communities.
- Though one of the most critical uses of traceability is ensuring that a design element satisfies a requirement, the existence of such a link may not answer the question: are the functionalities of the design element *required by* requirements? As a part of the validation and verification process, such a question should be answered to ensure that there are no unnecessary functionalities in a system component that are not driven by user needs; i.e., the links should be bidirectional to allow requirements tracing forward, from requirements to system components, and backward, from system components to requirements.
- As systems requirements evolve over the lifecycle, it will be beneficial to assess the impact of changing requirements. If the design segments *derived from* requirements and the implementation that *generated by* design segments are readily identified, the project manager will be able to make an informed decision about the effort needed to implement the required changes. A traceability scheme may provide both qualitative (e.g., the criticality of modules affected) as well as quantitative (e.g., the number of system components, code segments affected) to aid decision making.
- An important component of traceability information is design rationale. Rationale specify the *why* of decisions and trade-offs made throughout the systems development process. This information will be of interest to a variety of stakeholders who are interested in understanding, modifying and communicating decisions made throughout the system development process. Further, this information will be extremely useful in change management in the contexts of evolving requirements and assumptions. The

rationale for systems components could be *explicitly or implicitly specified* by requirements or could *result from* design decisions.

Hamilton and Beeby [16] define traceability as the ability to discover the history of every feature of a system. Design rationale is an important component of such a history. Brown [17] states that it should be possible to identify the requirement or design decision from which a product was derived. Design rationale identifies not only the decisions, but also the reasons behind them.

- Traceability information can be used in project tracking and management. Traceability links between various components of the system may include information used in project management (such as completion date, status, personnel assignment). Such an integration of project management as a part of the systems development process will ensure that timely and accurate information is available for critical project management tasks.
- As complex systems are composed of interdependent components, such *dependencies* should be represented and maintained. Often the inter-component dependencies are not well understood and documented.

Systems design is a complex activity involving interdependent decisions. In the absence of mechanisms to record such dependencies, over time and with changing development teams, this information will be lost. Such dependencies may span across different system components. A decision about software may be *dependent on an earlier decision* about hardware. For instance, a hardware decision to use SUN Sparcstations as the hardware platform may lead to a software decision that uses SUNOS as the operating system. As the system evolves over its life cycle, the hardware decision may get changed leading to inconsistency with the software that was based on the earlier hardware decision. Unless the dependencies are captured and maintained, such issues may go undetected leading to severe system integration problems. Our model will provide mechanisms to represent and reason with the dependencies among design decisions.

- A major use of traceability is the identification and assignment of accountability. Examples of such linkages include system components *designed by*, system components *tested by* or system components *validated/verified by* or *modified by* development personnel. Maintenance of accountability information will facilitate communication, coordination and maintenance of a system. It is especially important to maintain this information in mission critical areas of the system. An analogy will be similar the information maintained in aircraft construction and maintenance.
- A comprehensive mechanism for traceability should link the "humanware" component of a system to the other components of the system. Examples of such linkages include system functionalities *performed by* humans. This information is necessary to ensure that the allocation decisions are complete and correct.
- Traceability linkages (e.g., *explained by*) to systems components such as manuals, policies and procedures that specify how to obtain a required performance from a system component are as important as the information about the 'why' of the design process.

- Automated support for traceability is extremely important given the volume and the complexity of the task. Traceability information should be captured as a part of the systems development process automatically when possible. It is especially desirable and convenient to do so in situations where components are *derived from* or *equivalent of* (such as a decomposition) relationships (e.g., data flow diagrams and structure charts).

5 Design Rationale as an Example of Traceability

A conceptual model and mechanisms for the representation of and reasoning with process knowledge (i.e., design rationale) have been developed in earlier research as a part of the REMAP (Representation and MAintenance of Process Knowledge) project. The model and the mechanisms provided by REMAP for representing and reasoning with traceability information to support various stakeholders is discussed in detail elsewhere [18]. This design rationale model can be viewed as an instance of a traceability link between a requirement and a design element. The term "design element" denotes any part of the system design or implementation (i.e., data flow diagrams, specifications, pieces of hardware, humanware etc.). In this section, we discuss how such a model and reasoning mechanisms can be used in the context of the issues discussed in the previous section.

- support for various stakeholders: There are a variety of stakeholders involved in large software projects, each having a different set of goals and priorities. For each of the stakeholders, some useful support can be provided by recording in some structured manner, the history of a design in the form of (design) rationale.
- partially satisfied requirements: The process of satisfying requirements may generate several issues that need to be resolved. Resolution of issues lead to one or more design components. Partially satisfied requirements may be identified with unresolved issues that relate to that requirement using structures like the AND-OR graphs in REMAP. A similar structure can be used in linking design artifacts to requirements through design decisions.
- criticality of requirements: Our model captures the elaboration and refinement of requirements. Critical 'mission statements' or core 'business process' objectives can be the origin of such an elaboration and refinement. During this process, the criticality or importance of requirements can be ascertained and monitored. The REMAP model can represent this information as an attribute of the links between mission statements/business processes and requirements or as attributes of requirements themselves. Then, the critical requirements can be monitored to ascertain whether all the issues related to them are resolved in a timely manner.
- qualitative and quantitative reasoning: The strength or other characteristics of relationships can be either qualitative or quantitative. In REMAP, the contents of the primitives can be informal information (such as text). But the model has well defined semantics of relationships among its primitives, facilitating reasoning with this structure. For instance, the assumptions in a design situation can be given different degrees

of belief (or validity), and these beliefs can be automatically propagated to beliefs in arguments, positions and so on. Further, the strengths can be either qualitative or quantitative.

- **change management:** In REMAP, changes to design rationale will automatically trigger changes in the belief status (or validity) of design solutions thereby suggesting redesign [18]. Since various components of the process knowledge that lead to the design solution are tightly related, changes to the constraint set resulting out of changed assumptions, decisions or requirements will initiate the synthesis of a new design solution and provide rich information to estimate the effort involved in redesign.
- **project management:** REMAP provides facilities for representing and reasoning with temporal information which can be useful for project management. For instance, a validity time can be assigned to issues which could be interpreted as the time frame during which that issue must be resolved. Then, this information can be used for generating reminders to the designers or managers to focus their attention on issues that may have to resolved within a time frame or used in rank ordering issues. Project planning and control can be facilitated by integrity constraints on its primitives. An example of such a constraint could state that no requirement can be elaborated or refined until all requirements with higher priority or earlier validity time are considered.
- **accountability:** The REMAP environment facilitates the automatic capture or the representation of accountability information associated with design rationale.
- **Links to all system components:** The REMAP model can be used to capture relationships between requirements and all system components, including humanware, hardware, software etc.
- **automated support:** REMAP provides automated support for different stakeholders including interactive querying and updating of the design rationale knowledge base, a client-server architecture for multi-user support, a textual as well as hypertext-like user interface to the knowledge base and a reason maintenance system for maintaining and reasoning with design rationale.
- **derived links:** REMAP provides facilities for inferring knowledge based on deductive rules and facilitates the derivation of implicit links between requirements and design artifacts. For instance, a rule could state that if a design element is created by a decision, and the decision resolves an issue and the issue was generated by a requirement, then the design element traces to the requirement.

6 Conclusions

The preliminary analysis of the results of our initial study suggest that comprehensive models of traceability need to be developed. An approach to developing such models is to understand the traceability needs of various stakeholders in the systems development process. Further, a model of traceability should represent and reason with the semantics of various

traceability relationships in supporting system development and maintenance activities. Our current work has investigated the use of REMAP design rationale model and the reasoning mechanisms supported by it as an example of such an approach. Development of similar models and mechanisms to cover other important aspects of traceability are being addressed in ongoing research.

References

- [1] C. Potts and G. Bruns, "Recording reasons for design decisions," in *Proceedings of the 10th International Conference on Software Engineering*, (Singapore), pp. 418-426, April 1988.
- [2] S. Greenspan and C. McGowan, "structuring software development for reliability," *Microelectronics and reliability*, vol. 17, pp. 75-84, 1978.
- [3] N. Schneidewind, "Evaluation of secnavinst 3560.1 tactical digital systems documentation standard for software maintenance." Technical report NPS54-82-003. Naval postgraduate school, Monterey, CA 93943, February 1982.
- [4] N. Schneidewind, "software maintenance: improvement through better development standards and documentation." Technical report NPS-54-82-002. Naval Postgraduate school, Monterey, CA 93943, February 1982.
- [5] U. D. of Defense, "Military standard. defense systems software development." February 1988.
- [6] N. Walters, "Requirements specification for ada software under dod-std-2167a." *Journal of Systems Software*, vol. 15, pp. 173-183, 1991.
- [7] Teledyne Brown Engineering, Huntsville, Alabama, *requirements tracer (RT) user's manual*, October 1991.
- [8] S. Choi and W. Scacchi, "Assuring correctness of configured software descriptions," in *proceedings of the 2nd international workshop on software configuration management*, (Princeton, NJ), pp. 66-75, October 1989.
- [9] J. Baldo Jr., "Reuse in practice workshop summary." IDA Document D-751. Institute for defense analysis, Institute for defense analysis, April 1990.
- [10] E. Horowitz and R. Williamson, "Sodos: A software documentation support environment - its use," *IEEE transactions on software engineering*, vol. 12, pp. 1076-1087, November 1986.
- [11] E. Horowitz and R. Williamson, "Sodos: A software documentation support environment - its definition," *IEEE Transactions on Software Engineering*, vol. 12, pp. 849-859, August 1986.

- [12] M. Dorfman and R. Flynn, "Arts- an automated requirements traceability system," *The journal of systems and software*, vol. 4, pp. 63-74, 1984.
- [13] B. Ramesh and V. Dhar, "Design rationale capture and use in remap project," in *proceedings of the American association of AI-92 workshop on design rationale*. (San Jose, CA), AAAI, June 1992. To appear.
- [14] S. Wright, "Requirements traceability - what? why? and how?." in *Proceedings of the Colloquium on tools and techniques for maintaining traceability during design*. (Savoy Place, UK.), IEE, December 1991. IEE digest no. 1991/180.
- [15] W. Bryan and S. Siegel, "Making software visible, operational, and maintainable in a small project environment," *IEEE Transactions on Software engineering*, vol. 10, pp. 59-67, January 1984.
- [16] V. Hamilton and M. Beeby, "Issues of traceability in integrating tools." in *Proceedings of the colloquium on tools and techniques for maintaining traceability during design*. (Savoy Place, UK.), IEE, December 1991. IEE digest no. 1991/180.
- [17] B. Brown, "Assurance of software quality," SEI curriculum Module SEI-CM-7-1.1 (preliminary), Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA., July 1987.
- [18] B. Ramesh and V. Dhar, "Supporting systems development using knowledge captured during requirements engineering," *IEEE Transactions on Software Engineering*, June 1992. (To appear).

AN INTERACTIVE NATURAL INTERFACE FOR FORMAL CAPTURE OF COMPLEX SYSTEM REQUIREMENTS

Laura G. Hinton and Nicholas E. Karangelen

Trident Systems Incorporated
Fairfax, Virginia 22030

ABSTRACT

Specification of top level system requirements is the critical first step in the development of large complex computer-based systems. These requirements must be captured in a clear and unambiguous form to avoid unintended interpretations by design personnel and to permit expeditious pursuit of system development. This paper addresses the opportunity for application of embedded expert system technology in the development of a natural language interface which is tailored to support implementation of semi-automated environment for system requirements generation and capture.

INTRODUCTION

Specification of top level system requirements is the critical first step in the development of large complex computer-based systems. These requirements must be captured in as clear and unambiguous form as possible to avoid unintended interpretations by design personnel and to permit expeditious and direct pursuit of system development. Top level requirements for large complex systems must also be internally consistent and complete to support efficient development of system design and to avoid false starts due to incorrect or vague requirements. Top level system requirements are often subject to broad interpretation particularly if they do not include quantitative measures. English language statements of required functionality can be inherently ambiguous and may contain multiple meanings which are context sensitive.

The development of top level system requirements typically includes a broad spectrum of performance, functional, operational and other requirements established by domain experts who understand the customers needs and constraints. These domain experts often have considerable experience in the development and/or operation of similar systems but may not have formal training or experience in the use of formal or semi-formal specification techniques (i.e. essentially all top level specifications are written in english).

Natural language interface techniques potentially provide a mechanism for these domain experts to interactively employ a semi-automated tool and a structured method which supports system requirements generation and capture using common english which would require little or no training. Expert system methods represent a powerful approach in the realization of natural interface for requirements capture and analysis. This paper addresses the opportunity for application of embedded expert system technology in the development of a natural language interface which is tailored to support implementation of semi-automated environment for system requirements generation and capture. The interactive natural language interface is intended to act as a guide to domain expert users in development of formal, consistent, unambiguous requirements for large complex systems.

NATURAL INTERFACE OBJECTIVES

One of the principal objectives for any natural interface is to reduce the effort expended by an operator in learning and using a given operator-machine interface. In addition, this proposed natural interface for top level requirements capture will provide a mechanism for informal unstructured english text to be converted to a set of formal structured requirements through an interactive session with an expert system conducted in english. The formal structured requirements captured using this approach can also be converted to an equivalent statement of requirements in english.

The payoff in implementation of an effective natural interface is greater where the user's time is highly valued (e.g., senior executives) and where familiarity with machine oriented interfaces is low (e.g., many domain specific experts). Domain experts who typically create these top level system requirements must also be the ones who refine them, or the requirements might be misinterpreted in the refinement process. An interactive natural interface to a formal structured requirements capture method would provide both ease of use for the user and a well structured, clear requirements specification product.

The natural interface concept described in this paper supports generation and capture of a clear, unambiguous, complete statement of the top level requirements and design constraints for a complex system. The interface is intended to provide uninitiated users who are subject matter experts a mechanism for developing and capturing effective complex system top level requirements. The natural interface methodology should accommodate different perspectives of the system or the system requirements which may be held by various domain experts in specific areas (i.e., the various engineering specialties).

NATURAL INTERFACE AND EXPERT SYSTEM EMPLOYMENT STRATEGY

The symbolic processing technique represented by a forward-chaining inference engine is a natural match to implementation of a natural structured interactive english interface and to the pursuit of generating and refining a complete unambiguous set of top level system requirements. The essential strategy for implementing an expert system-based natural interface for requirements capture is to combine a high performance graphic interactive operator machine interface with an innovative expert system designed to conduct an interactive dialogue with the user through natural english language constructs.

The expert system creates a dialogue with the user implementing an initial rule set based on the english language, the formal structure, and information specific to the project's domain. Clarification of meaning are resolved by the user, and the expert system "learns" based on the user responses to the expert system questions. For example, if a group of words are categorized as domain-specific "jargon" in the first paragraph of a sentence, that jargon is recognized throughout the rest of the paper. The ability of the expert system to "learn" and "remember" supports the capability of the interface to parse through a document without using complicated language recognition techniques.

The expert system provides a mechanism for natural interactive exchange with the operator, and guides the capture of requirements to ensure consistency, avoid ambiguity, and to help achieve completeness based on a format representation. To help the user better understand the formalized requirement, a structured english version is created. This Equivalent Statement of Requirements (ESOR) is a limited english explanation of the formal requirements. The ESOR allows domain experts to understand and examine the formal requirements and ensure that the converted requirements are consistent with their intentions.

PROTOTYPE SYSTEM REQUIREMENTS CAPTURE TEST BED

A rapid prototype test bed for examining the employment of expert system technology to top level system requirements capture and analysis has been created using UNIX and the X Window System on a Sun SPARCstation 2, and is written in the C programming language. The expert system employed is CLIPS (NASA sponsored through COSMIC) which is embedded in the tool and fully integrated within a UNIX process. A preliminary set of rules have been developed and demonstrated which address the three key capabilities of the prototype system: (1) parse english language requirements statements, (2) generate a formal structured statement of the requirement in the form of an Information Model, and (3) ask the domain expert user questions to support correct parsing of the requirements and to provide additional data to complete the Information Model. The user interacts with the expert system by responding to questions and asserting additional information. The expert system leads the user through a process designed to generate consistent unambiguous requirements statements through natural english interaction with the user.

A unique implementation of CLIPS has been developed as part of the prototype test bed. This implementation introduces the concept of "metarules" as they pertain to natural english language requirements parsing and creating a structured formal representation of these requirements. This approach is discussed below.

ENGLISH LANGUAGE REQUIREMENTS PARSING AND A METARULE APPROACH

Parsing english language requirements and creating a structured formal representation of those requirements (such as an Information Model) through an interactive session between operator and machine is accomplished using a forward chaining expert system approach. The objective is not to extract the semantic meaning of an english sentence, which is a much larger and potentially overwhelming task, but to guide the human operator through a process of requirements refinement and capture. The natural interface proposed here to support system requirements capture focuses on a carefully chosen subset of english grammar and vocabulary which is common to the majority of top level requirements. The selected subset is tailored to provide a robust english language communication capability and to limit the processing required to parse and understand the user input.

The embedded expert system is constructed in a way to maximize the flexibility of the knowledge base through the use of an innovative metarule approach. The metarule approach employed is described in the following paragraphs which begin by describing the typical operation of a forward chaining inference engine and then contrast that operation with the metarule concept. The advantages and disadvantages of the metarule approach are also addressed.

An expert system is typically composed of an inference engine and a knowledge base. Within the knowledge base is one or more sets of rules and facts. Typically the rule base is fixed for a given execution of the inference engine (although different subsets of rules may be employed at various times) and the facts change as a function of external inputs and the operation of the inference engine. In the case of a forward chaining inference engine the rules are constructed in an "if a then b" format where a is the antecedent fact and b represents a consequent fact. (Both a and b can represent complex boolean expressions of antecedent facts or consequent facts respectively.) The inference engine looks for matches in the current facts with the antecedents of the rules. When a current fact matches an antecedent of a particular rule, the rule is said to fire by essentially asserting the facts in the consequent side of the rule. The new facts are then addressed in the same manner by the inference engine until no additional facts are generated.

The metarule approach to knowledge base creation and maintenance as applied in this research employs rules and facts in a somewhat different manner. In this unique implementation of CLIPS, domain-specific rules are pre-processed and represented in CLIPS as facts. The CLIPS rules that exist in the knowledge base (referred to as the metarules), consist of four format-dependent rules that operate on all of the user-defined, domain-specific rules (see Figure 1). Using this approach, the expert system partition of this system is designed to efficiently process those aspects of a natural interface language parsing to support system requirements capture.

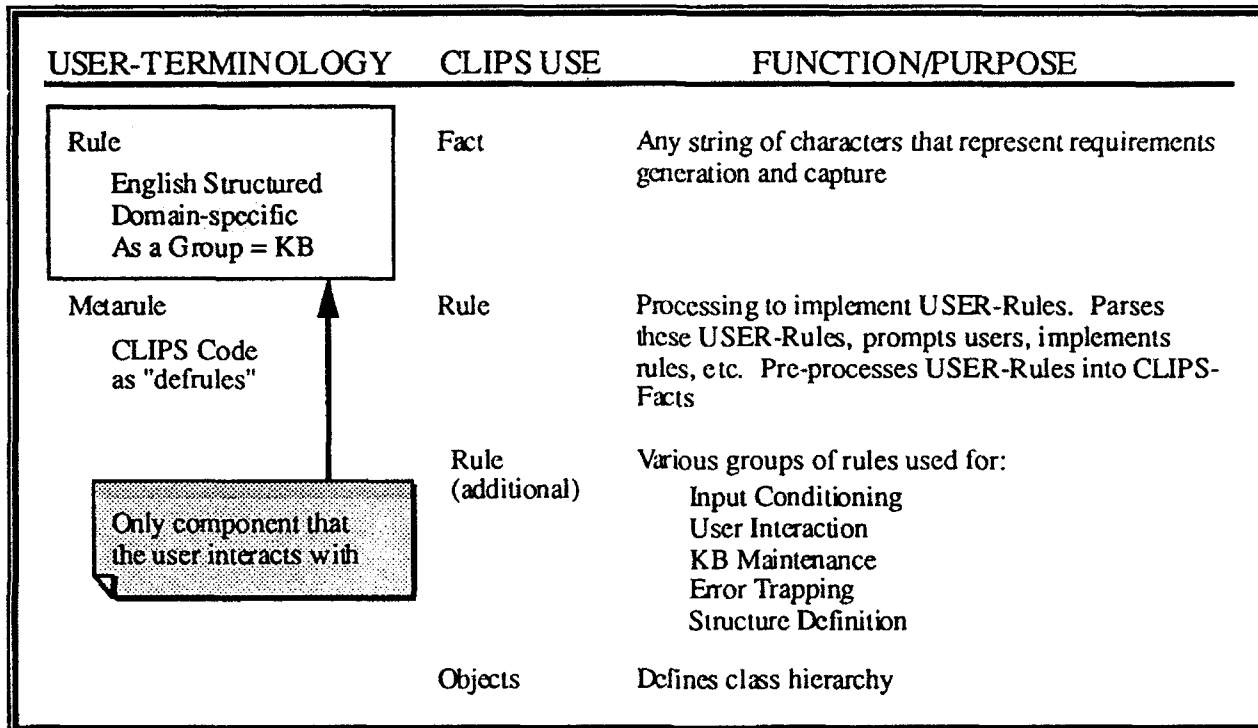


Figure 1: CLIPS Metarule Approach

Metarules are designed to operate with depth-first conflict resolution and forward inferencing. *The term "metarule" refers to any CLIPS rule which is used to process another rule.* Metarules can only operate on rules that exist in a predefined form. The metarules "Detect a Valid Antecedent Clause", "Conjunctive Antecedent Clause", "Execute User Rule", and "Mutually Exclusive Parameters" are supported by:

<valid clause>	\\	formats used by the metarules
ManageRule Function	\\	communication from C++ to CLIPS
<result>	\\	communication from CLIPS to C++

Information is represented in the CLIPS-based expert system through facts, objects, and global variables. The user has access (modification, authoring, examination) to the sets of "if-then" rules and their associated facts. During run time, facts/rules and metarules are maintained in the knowledge base, which is processed by the inference engine. Although the knowledge base can be accessed in a number of ways, it is processed only by the inference engine. The result of this processing is modification of the knowledge base and creation of additional facts.

To emphasize, *all user-defined rules are internally stored as pro forma "facts" in the knowledge base*. The metarules are able to identify these facts and to process them as logical decisions of the sample type:

Whenever <antecedent> then <consequent>.

Where:

- <antecedent> represents one or more preconditions
- <consequent> represents one or more actions which result if all antecedents are satisfied. Consequents can act as antecedents in other rules (i.e. rules can be chained).

The advantages of this metarule approach include processing speed, and simplicity of user implementation. Bench mark tests on using CLIPS as an embedded system indicate a dramatic increase in processing time with respect to the number of rules that the inference engine processes. Tests run on number of rules, versus number of facts showed conclusively that a small number of rules operating on a large number of facts is the optimal utilization of CLIPS. Simplicity of user implementation is another key advantage. From the user's perspective, the rules will be seen as english-like sentences and will be constructed primarily by responding to prompts from the user. The technique minimizes natural language processing, provides the flexibility of adapting to the user's lexicon and, most importantly, does not require the user to "program" rules in any particular order.

The potential shortcoming of the proposed metarule approach is the possibility that processing speed may still be slow. As CLIPS is an interpretive language (as opposed to compiled), execution speed is affected, even with the time-saving advantages taken by the metarule approach. However, using the expert system partition of this system as designed (to efficiently process those aspects of a natural interface language parsing to support system requirements capture) tend to be bound more by the graphic interface than with anything else, and is not anticipated to present an unsolvable problem.

PRELIMINARY DEMONSTRATION - EXAMPLE UHF RADIO REQUIREMENTS ANALYSIS

The following is a short example requirement which was processed using the prototype system. The example top level requirement is part of an actual unclassified UHF radio system Tentative Operational Requirement (TOR). The session begins by loading the raw (e.g. not analyzed) requirements document and initiating the expert system. The expert system then leads the user through an analysis of the requirements in a step wise fashion and builds a fact base and an Information Model as a result of the interaction with the user. As the fact base grows through interaction with the user, it is used by the expert system to maintain an internally consistent set of requirements and terms of reference. During the process the expert system constructs a formal structured Information Model (entity-relationship diagram) which captures the requirements as refined through the interaction and also develops an equivalent statement of requirements which is a structured english language representation of the Information model. Figure 2 illustrates the original requirement as input into the system as well as the Information Model and ESOR generated based upon an interactive session with the user.

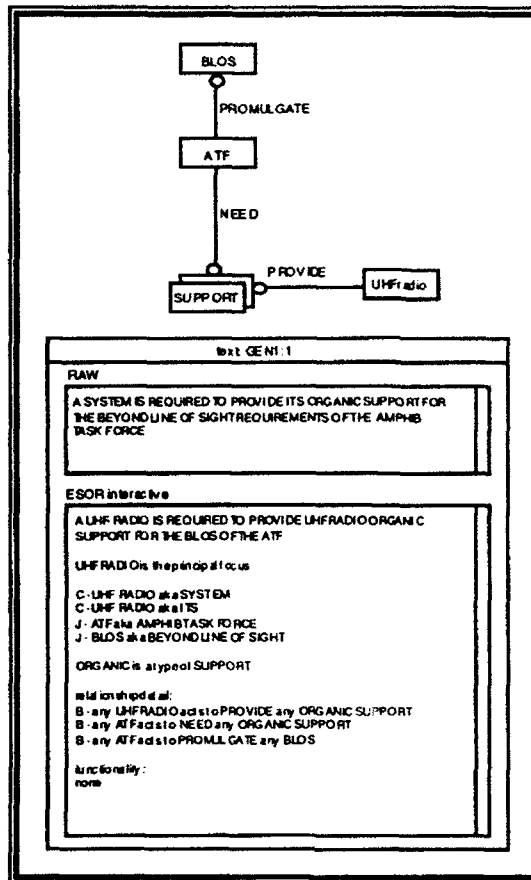


Figure 2: Prototype System Output

FUTURE RESEARCH

Continuation of this work will begin with identifying one or more formal structured requirements capture techniques which can serve as test cases for implementation of a full scale natural language interface capability. Research in natural language requirements parsing and development of rule based strategies for interactive creation of structured top level requirements will be influenced in part by the format and characteristics of the selected formal requirements structure. Additional aspects of the future work will include characterizing the english grammar and usage nominally associated with requirements specification, and development of graphic interactive techniques to enhance user interaction with both the expert system and the captured requirements.

ACKNOWLEDGEMENTS

This paper contains concepts initiated in an SBIR Contract under the direction of Steven Howel of the Naval Surface Warfare Center, Dahlgren Division (NSWCDD) White Oak, Maryland. This work also reflects concepts developed and refined in cooperation with the Systems Synthesis Specification Working Group (SSSWG) at NSWCDD. The authors would like to recognize the contributions of Mike Edwards, Steven Howel, and David Britton. The Office of Naval Technology (ONT) and Trident Systems Incorporated have provided support for this and related research by the authors.

REFERENCES

1. *CLIPS Architecture Manual (Draft Copy)*. CLIPS Version 5.1. Software Technology Branch, Lyndon B. Johnson Space Center. November 1991.
2. *CLIPS Reference Manuals: Volume I Basic Programming Guide, Volume II Advanced Programming Guide, Volume III Utilities and Interfaces Guide*. CLIPS Version 5.0 Software Technology Branch, Lyndon B. Johnson Space Center. January 1991.
3. Chorafas, Dimitris N. *Systems Architecture & Systems Design*. New York: McGraw-Hill Book Company, Inc., 1989.
4. DeMarco, Tom. *Structured Analysis and System Specification*. Englewood Cliffs, NJ: Yourdon Press. 1979.
5. Gause, Donald and Gerald M. Weinberg. *Exploring Requirements: Quality Before Design*. New York: Dorset House Publishing, 1989.
6. Giarrantano, Joseph C. *CLIPS User's Guide Volume 1 Rules*. NASA Lyndon B. Johnson Space Center, Information Systems Directorate, Software Technology Branch, Version 5.1. September 1991.
7. Giarrantano, Joseph C. *CLIPS User's Guide Volume 2 Objects*. NASA Lyndon B. Johnson Space Center, Information Systems Directorate, Software Technology Branch, Version 5.0. May 1991.
8. Hayes-Roth, Frederick, Donald A. Waterman, and Douglas B. Lenat, ed. *Building Expert Systems*. Reading, MS: Addison-Wesley Publishing Co., Inc., 1983.
9. Hu, David. *C/C++ for Expert Systems*. Portland, OR: Management Information Source, Inc., 1989.
10. *Integrated Computer-Aided Manufacturing (ICAM) Architecture Part II Volume V-- Information Modeling Manual (IDEF)*, Report No. AFWAL-TR-81-4023 Volume V, Materials Laboratory (AFWAL/MLTC)/AF Wright Aeronautical Laboratories (AFSC), Wright-Patterson AFB, OH 45433, June 1981.
11. Kimbrell, Roy E. "English Recognition", *Byte*, pp.125-140, December, 1985.
12. Pedersen, Ken. *Expert Systems Programming*. New York: John Wiley & Sons, Inc. 1989.
13. Schmuller, Joseph and Yingsheng Mi. "Expert System Shells at Work," *PC AI*, Vol. 5, No. 1, pp. 40-49, January/February 1991.
14. Van Horn, Mike, and the Waite Group. *Understanding Expert Systems*. New York: Bantam Books, 1986.
15. Wasserman, Anthony I. and Peter A. Pircher. "Object-Oriented Structured Design and C++," *Computer Language*, Vol. 8, No. 1, pp. 41-52, January 1991.
16. Waterman, Donald A. *A Guide to Expert Systems*. Reading, MS: Addison-Wesley Publishing Co., Inc., 1986.

INTEGRATED SYSTEM DESIGNER (ISD)

Mary Blanchard
Science and Technology Associates, Inc
Suite 700, 4001 North Fairfax Drive
Arlington, Virginia 22203

What information is necessary to represent complete system designs? How should this information be represented to support evolving designs? Can this representation support reusability? What format would enable all the engineers, end-users, and customers to view the system through the "same eyes"? How can the engineers ensure the proposed design fulfills the defined requirements?

While examining each of these questions, we explored the currently available CASE and CBSE tools. Each of today's tools provides a partial solution; what is lacking is a single representation addressing all the above questions. Our response is The Integrated System Designer (ISD) — a single method of representing systems which supports all the phases of product life cycle; domain analysis, system design, trade-off analysis, development, and maintenance. The proposed solution supports an iterative approach to the previously outlined tasks. Diagrams 1-3 summarize ISD and its relationship to existing tools.

Returning to figure 1, you will note many sources of both input and output. The seven system default libraries are one example. Each of these libraries will be predefined and include classes encapsulated with both attributes and services, from which users can create their own system representation. ISD will provide support for updating and expanding these libraries. Their existence serves two purposes: one, the user is able to quickly create designs, and two, the system has a systematic method for acquiring the details of the system design. This aids in both providing additional supporting documentation and a consistent set of input data to the analysis and simulation tools.

Actually this information is passed to the Automated Model Builder before any analysis or simulation. The Automated Model Builder is responsible for synthesizing all the information contained within the system design and creating an equivalent model that can be understood by the analysis and simulation tools.

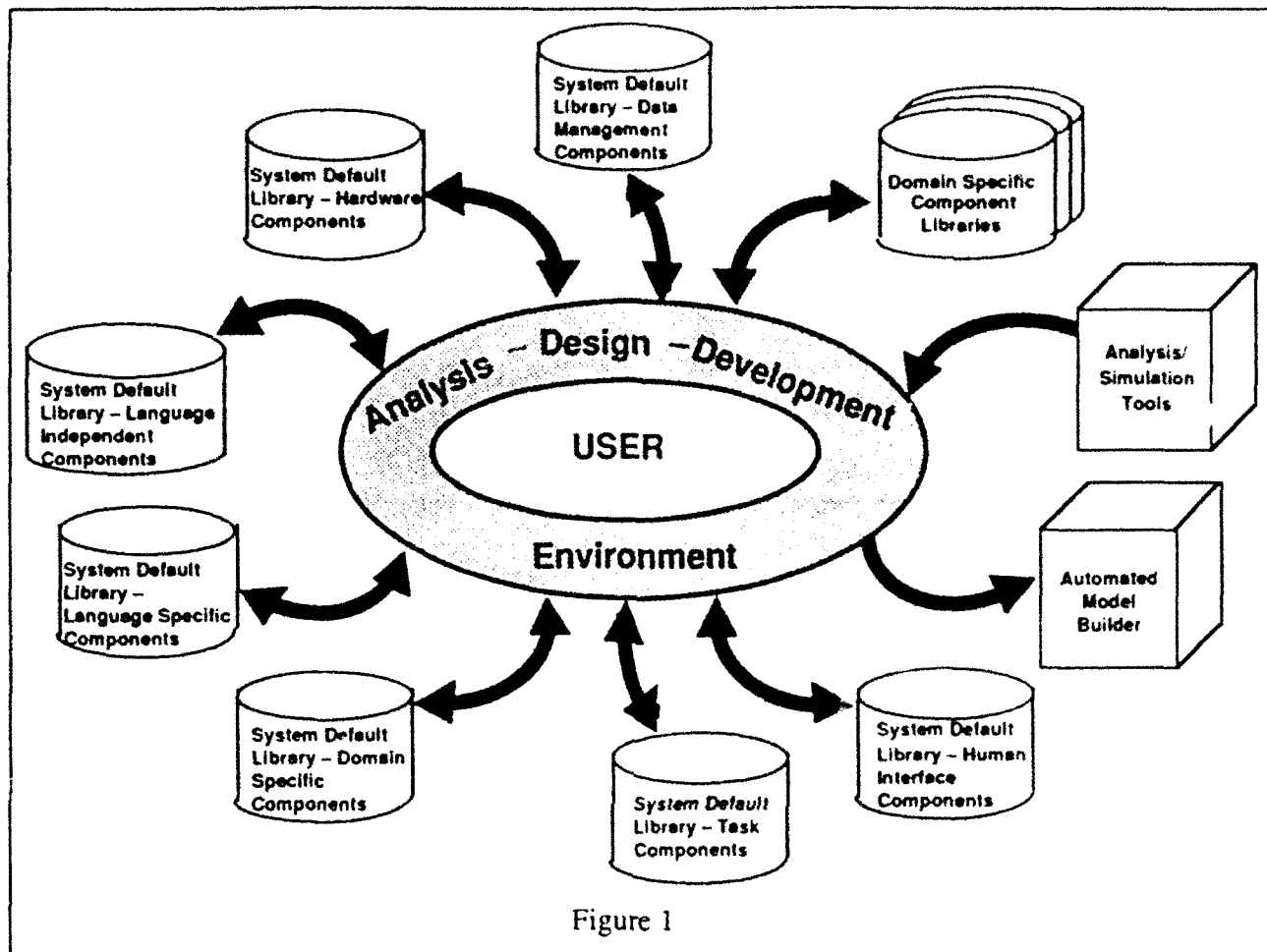


Figure 1

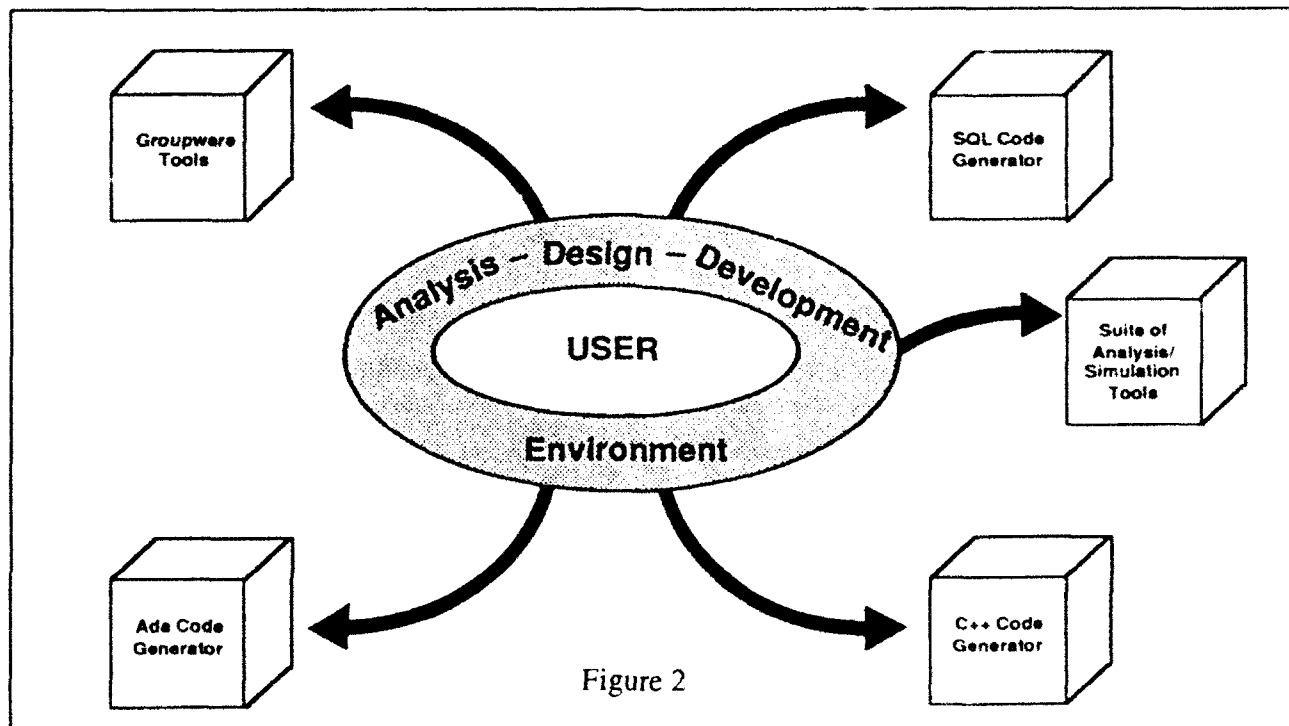


Figure 2

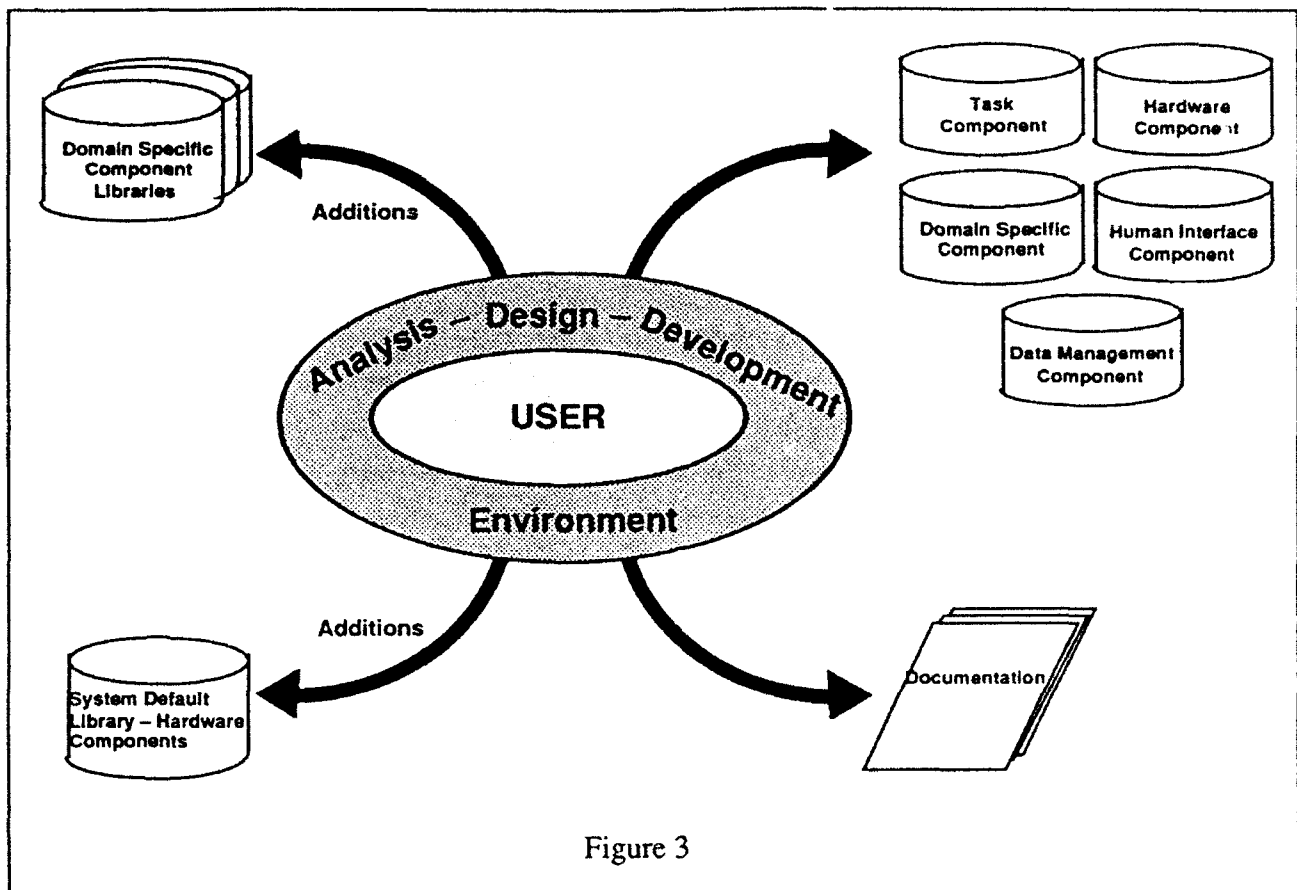


Figure 3

The other visible sources of input and output are the domain specific component libraries. Their usage is primarily for constructing the domains specific component of a new system, using components from past systems and applications. These components may range from complete applications to individual low-level classes. It is up to the user to select the appropriate level of complexity for reuse.

The components contained in

figure 3 represent the resulting system design. As you will notice the design is broken into five parts — domain specific, human interface, data management, hardware, and task. How a system is represented using this technique is explained in the next set of diagrams 4-8 and in the subsequent paragraphs.

Contained within the Domain Specific Component will be the people, physical entities, and abstract entities that exist within your domain. The Human Interface Component will represent those entities and operations that constitute the user interface. Tradition-

ally, these are menus, menu items, graphic icons, windows, and panes. However, your system may use a thermostat as an input device — whatever entities and operations your human interface consists of, represent it here.

Your data layout, which includes tables, files, and databases are represented in the Data Management Component. Each class and correspondingly, each object will have an associated set of attributes defining the layout and location of the data. The exact format will vary depending upon the type of data management scheme selected for your system design. Your design may also involve more than one data management scheme. Regardless of the scheme chosen, system default classes, with encapsulated attributes and services, will exist for you to use in building your layout. Associated with each data management scheme, will be an “object-server” class, with the services “store” and “retrieve”. In actuality, these two services will contain calls to other store and retrieve services, to account for each of the possible locations the data could be stored. This enables the analyzer and simulator to model storage locations such as cache and RAM, besides a secondary storage device.

Store all hardware devices, including CPUs, sensors, actuators, operating systems, networking components,

pumps, secondary storage devices, and any other physical devices within the Hardware Component. System default components with associated attributes and services will be available to the user to quickly assemble the design of the hardware system.

All the services defined within the Domain Specific Component, Human Interface Component, and Data Management Component will be mapped to both hardware devices contained within the Hardware Component, and human resources contained within the Domain Specific Component, in the Task Component.

The justification for this representation is based upon the history of the system design process. Historically, components such as the user interface has been very volatile, while components related to the specific domain have been more stable. Using this knowledge we have split the design into its respective volatile and stable components. Since system performance is one of our concerns, it is critical to isolate the hardware component from other system components. This enables the user to more easily test various hardware configurations, without significantly impacting the other components of the design.

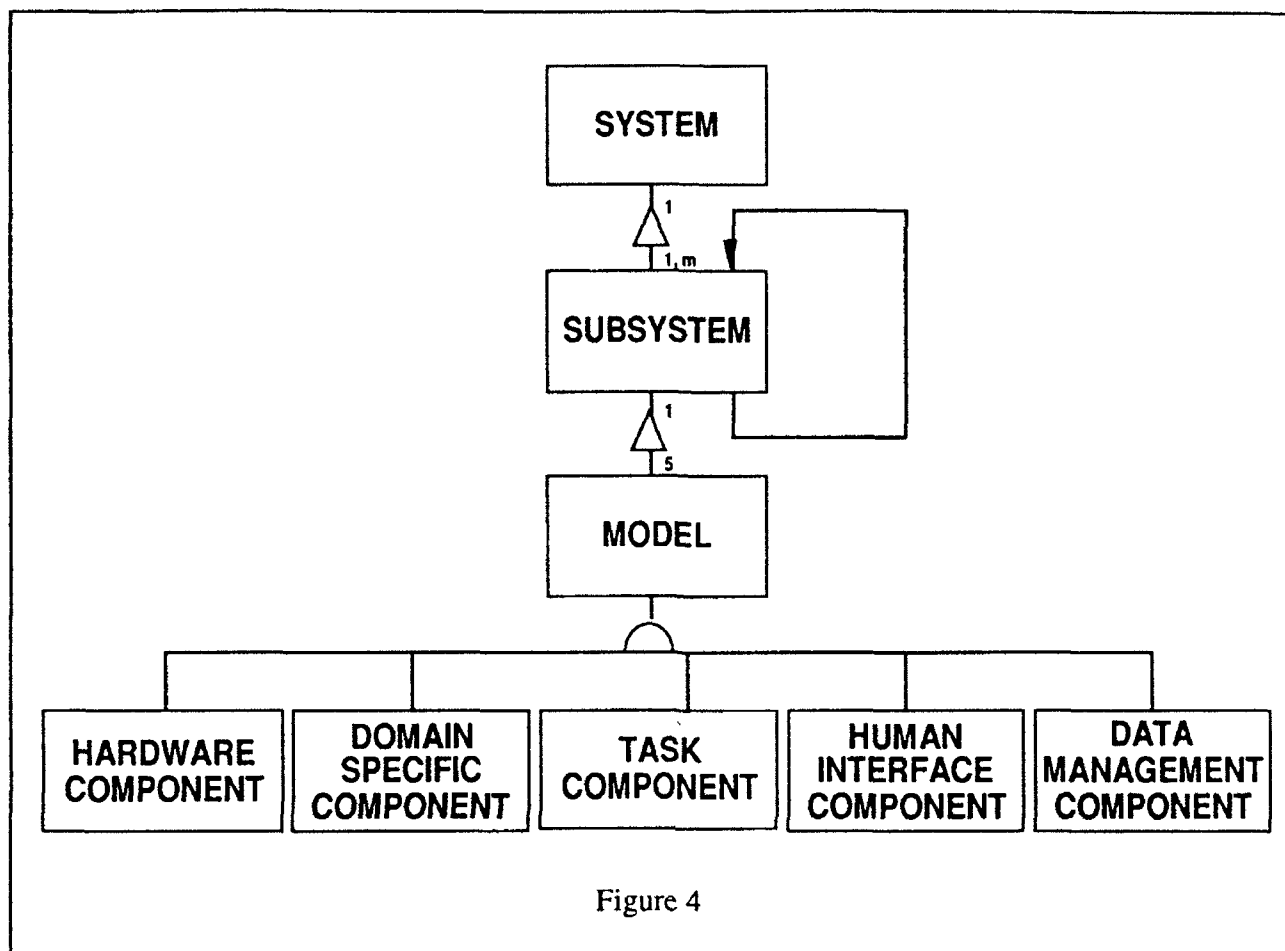


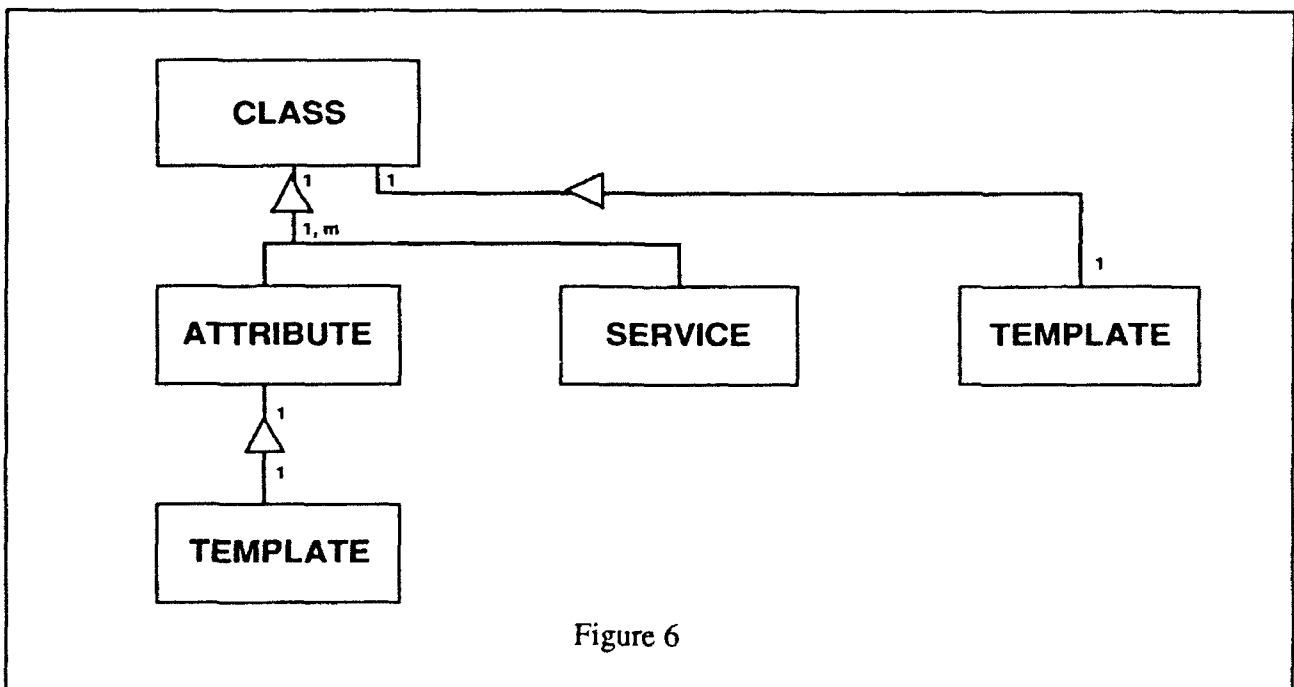
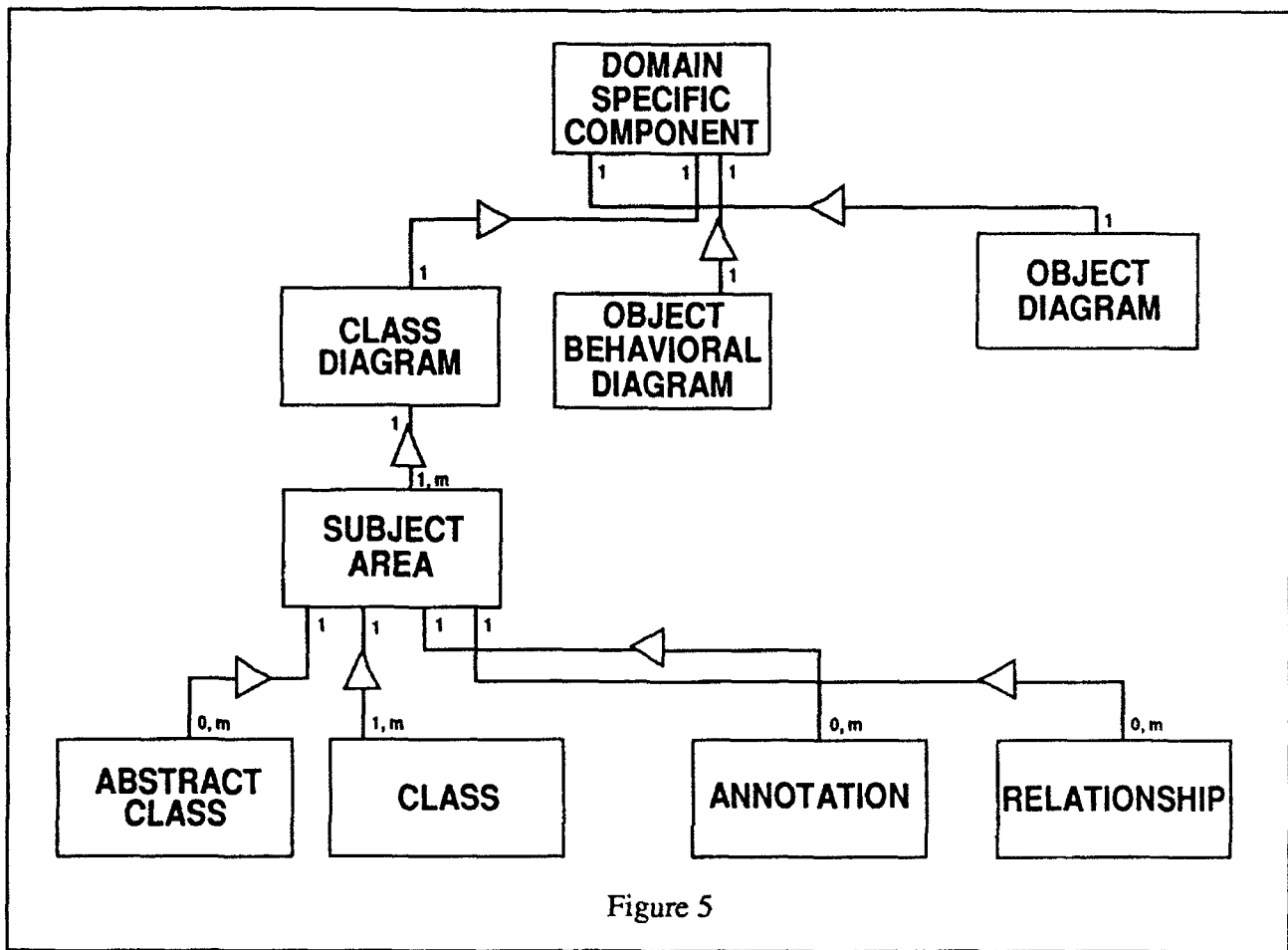
Diagram 4, illustrates the decomposition of a system, hierarchically from a top-down approach. ISD also provides support for both bottom-up and integrated approaches.

In diagram 5, the decomposition of the Domain Specific Component, is demonstrated. All the diagrams shown here are applicable to all five components — the exception being the “Object-behavioral Diagram”. This diagram is not applicable to the Task Component, since this information is already contained within the PDL description

of a task.

Diagrams 6 and 7, contain information typically associated with a class. Note, the rectangle containing “analysis” is not solid, like the others. The reasons being analysis results typically provide information corresponding to the entire design, not an individual class.

ISD supports the class relationships demonstrated in diagram 8.



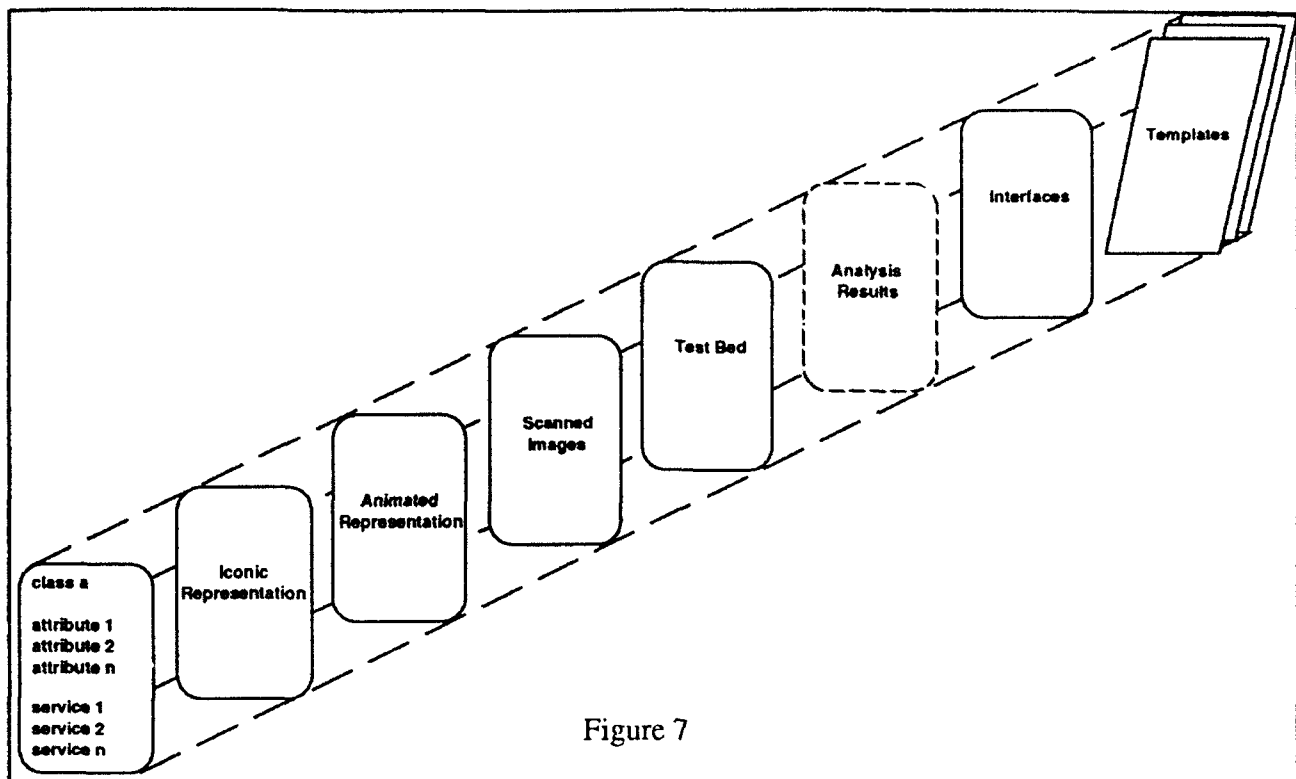


Figure 7

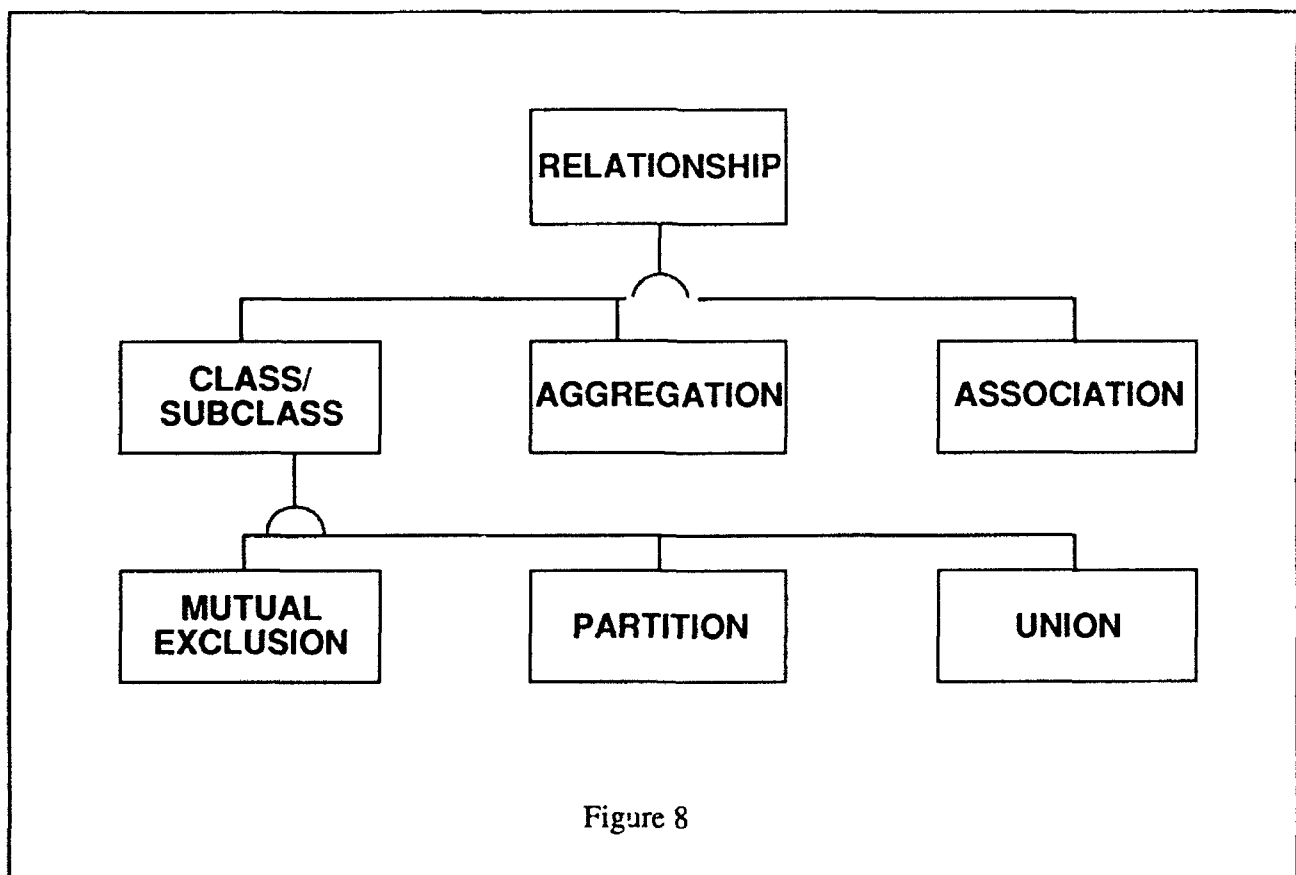


Figure 8

As will become apparent, our notation describes classes with their encapsulated attributes and services, associations, and hierarchical and aggregation relationships. Supporting templates are provided for each class, attribute, service and relationship. ISD describes objects in terms of the attributes, services, and relationships associated, with the originating class.

Additionally, objects are described in terms of the messages they send, in response to events and changes in state. The object-behavioral diagram provides the additional supporting notation required to describe an object's reactions to events and state changes. The subsequent diagrams 9 - 14, provide descriptions of the supporting constructs.

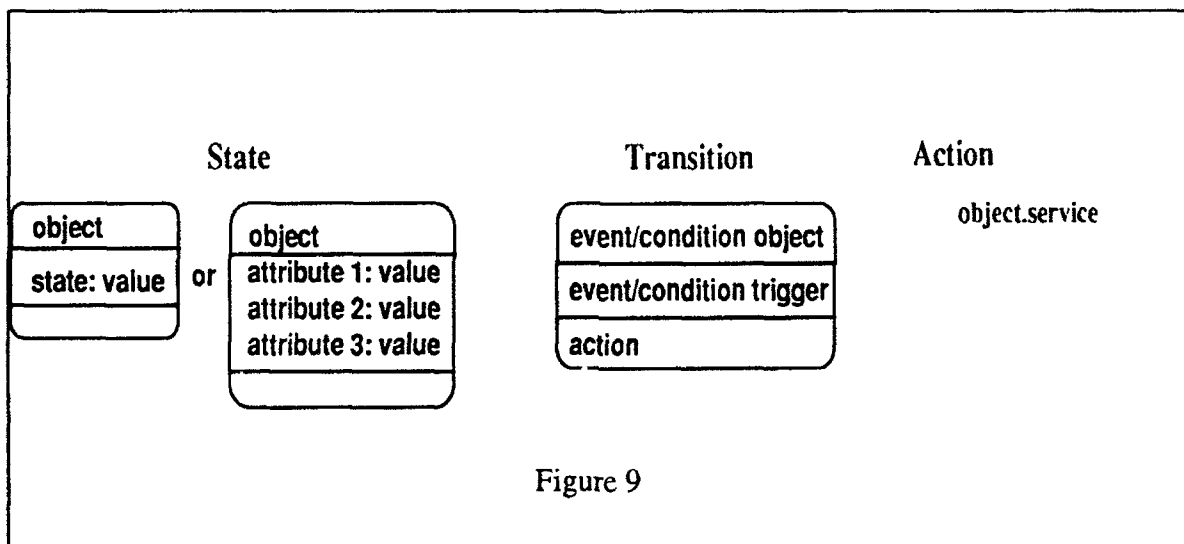


Figure 9

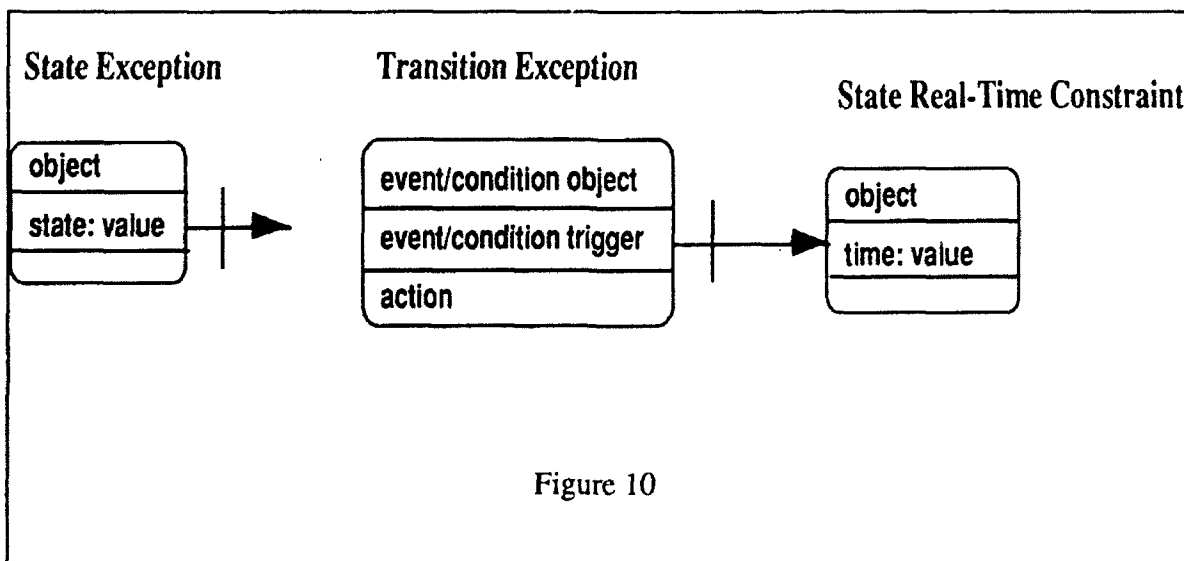
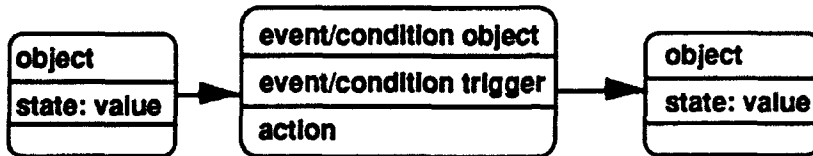
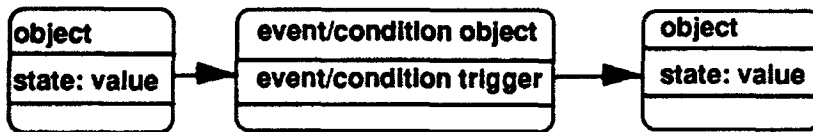


Figure 10

State – Transition with action



State-Transition without action



State-Transition with action, without event



State-Transition without action, without event

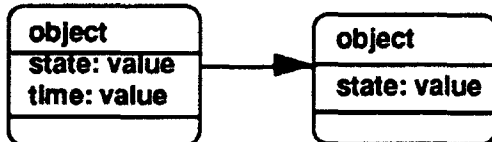
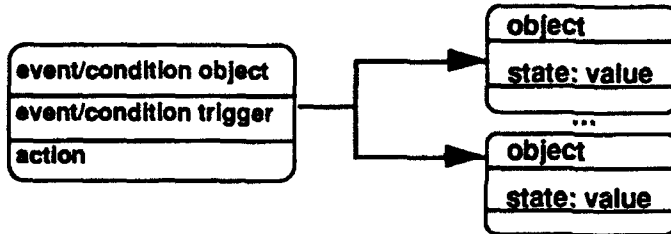
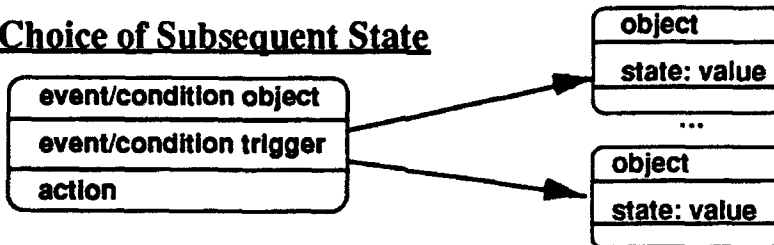


Figure 11

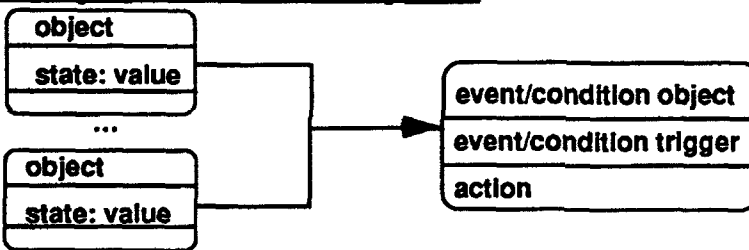
Entry into Multiple Subsequent States



Choice of Subsequent State



Multiple Prior States Required



Choice of Prior State

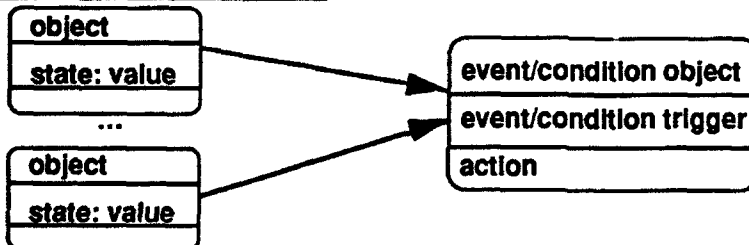
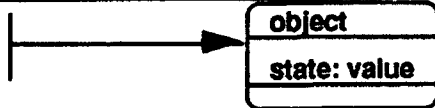
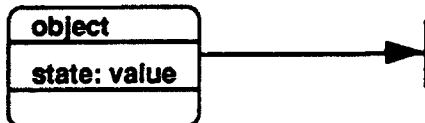


Figure 12

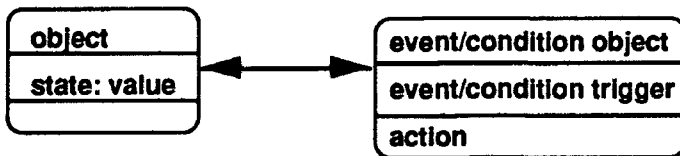
Initial Transition/Initial State



Final State/Final Transition



Return to Prior State



Remain in Prior State

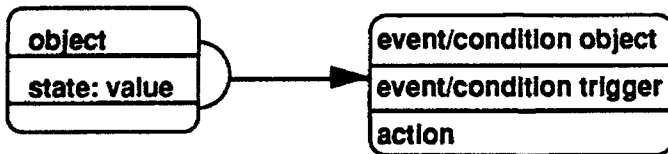
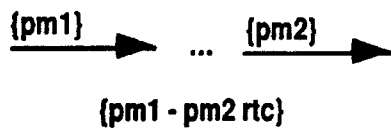
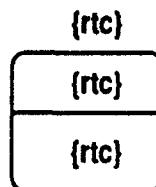


Figure 13

Path Real-Time Constraint



Transition Real-Time Constraints



rtc: real-time constraint

pm: path marker

Figure 14

In order to support top-down composition, it is necessary to support both class and service decomposition. Class decomposition is fairly straightforward and has been practiced for several years, as has functional decomposition. What is different is the requirement for the system design process to function in an object-oriented mode first, and then in a functional mode. This requires a slightly different process and representation than what has been traditionally used. As you will see in the following diagram, the user

begins by defining the appropriate classes at level n and decomposing these classes into classes at level $n - 1$. Beginning at level $n - 1$ and working up the hierarchy is also appropriate. Next, the user maps the services associated with level n , to an appropriate set of services at level $n - 1$ (see figure 15).

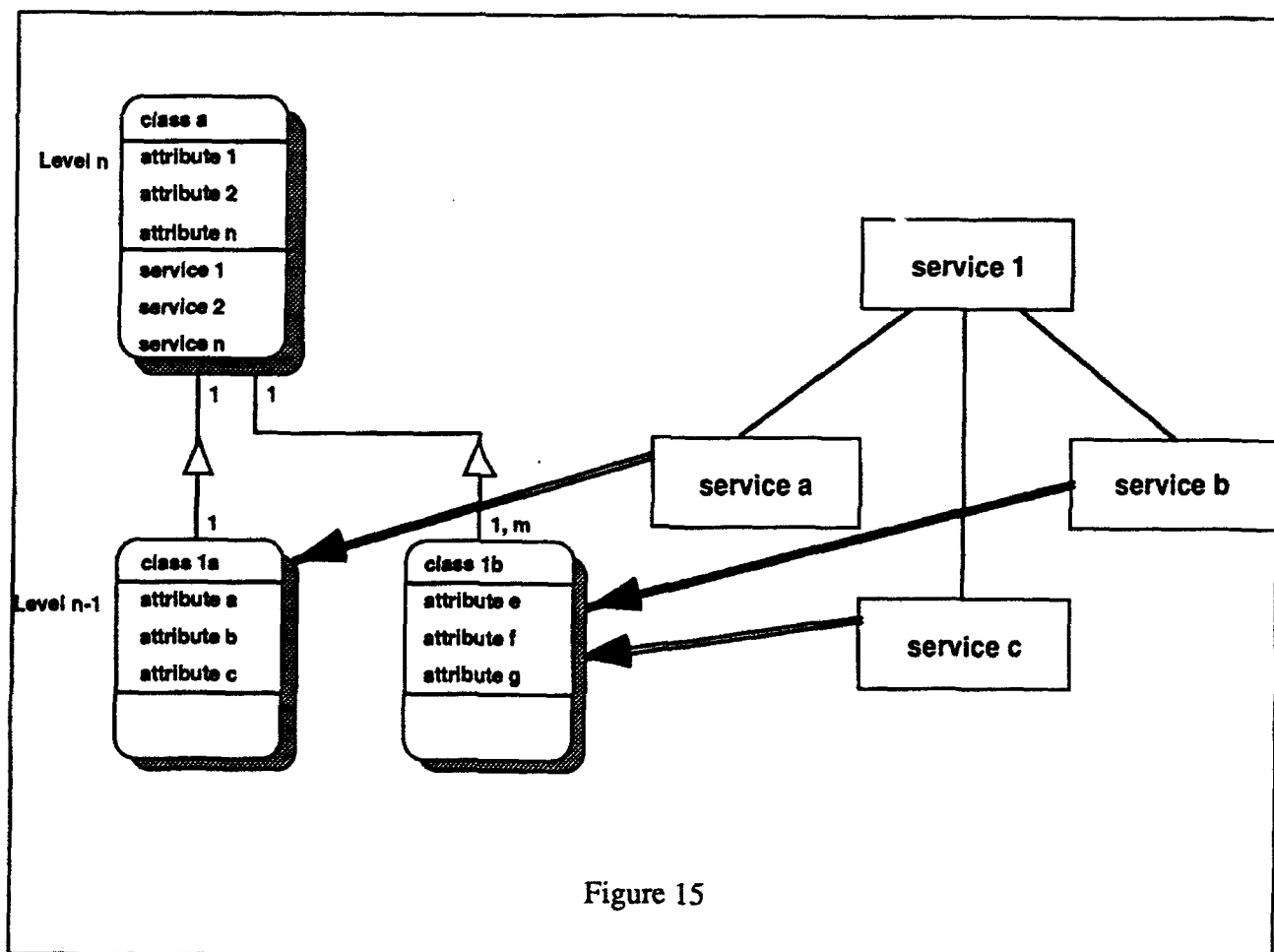


Figure 15

Each service is described using PDL. Before a language choice is made, language independent PDL is used; after a choice is made, the previous PDL descriptions are mapped to their corresponding language dependent representations. In many cases, the user will be asked for additional information. Typically, this additional information consists of attributes, associated

with language-specific components. Diagram 16 illustrates sample language dependent components, language independent components, and PDL constructs.

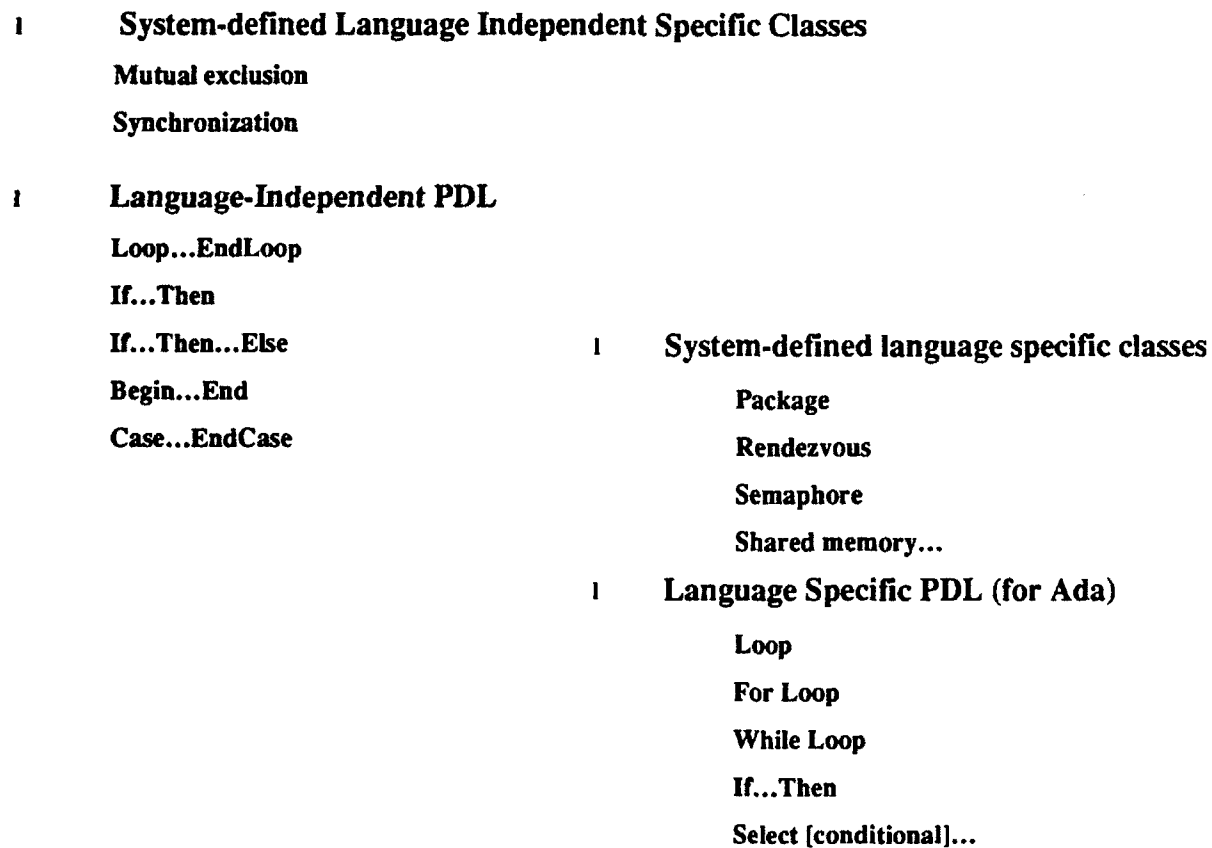


Figure 16

The Automated Model Builders uses the resulting system design in conjunction with the Building Blocks Table to produce the model for analysis and simulation. Understanding the model's representation is necessary to understand how the model is created. Figures 17 - 19 contain a sample of a model ready for analysis and simulation.

It appears to be a long series of if-statements — which it is. However, associated with one of the if-statements is an “event”. This “event” translates to a transition within the analysis or simulation tool. This “event” translates to a service on the system description side of the tool. Also, on this diagram you will see probabilities. These represent the frequency of choosing one path over another in the system design. Other data includes the amount of time required to execute the service or service statement corresponding to the “event”. The execution time is assumed to be for a non-preemptive scenario. Preemption is modeled through additional constructs created by the automated model builder and by the “queuing up of data” at a resource.

The system produces models by first examining each of the resources contained within a system design, the operating system (if applicable) executing on each of these resources, and the tasks executing on each of these re-

sources. Additionally, information about each of these components, as well as PDL descriptions of tasks and services are used to locate and complete corresponding constructs within the Building Blocks Table (see figures 20 - 23). The Automated Model Builder attaches each newly created construct or set of constructs to the “model under construction”. These steps are repeated until the list of resources, within the system design, are exhausted. At this point, the model is complete and ready for analysis and simulation.

Figure 24 documents the results of the analysis and simulation processes.

Work on both the Building Blocks Table and the system analysis/design representation is on-going. The focus is currently on testing the analysis/design representation on as many different types of systems as possible. Future work includes building up the “Building Blocks Table” to support many different designs.

If	event physical device clock-C1.interrupt
Then	suspended queue, priority 0 ready queue, priority 0
If	CPU SDP available
Then	ready queue, priority 0 selected ready queue, priority 0
If	CPU SDP available
Then	ready queue, priority 1
Unless	selected ready queue priority 1 ready queue, priority 0
If	task task SDP-clock, priority 0
Then	selected ready queue, priority 0 operating system-SDP
If	operating system-SDP
Then	task task SDP-clock, priority 0 START task running

Figure 17

If	event physical device clock-C1.interrupt
Then	suspended queue, priority 0 ready queue, priority 0
If	CPU SDP available
Then	ready queue, priority 0 selected ready queue, priority 0
If	CPU SDP available
Then	ready queue, priority 1
Unless	selected ready queue priority 1 ready queue, priority 0
If	task task SDP-clock, priority 0
Then	selected ready queue, priority 0 operating system-SDP
If	operating system-SDP
Then	task task SDP-clock, priority 0 START task running

Figure 18

If task SDP-clock.if clock-C1.compare_ticksize_interrupt count = true
 task running
Then task SDP-clock idle
 task running
 CPU SDP available
 suspended queue, priority 0
 Probability associated with first condition of If-statement: 0.8

If task SDP-clock.if clock-C1.compare_ticksize_interrupt count = true
 task running
Then wait 3
 task SDP-clock. clock-C1.send signal to calendar requested.
 task running
 Probability associated with first condition of If-statement: 0.2

If wait 3
 task SDP-clock.clock-C1.send signal to calendar completed
Then task running
 cpu SDP available
 suspended queue, priority 0

Figure 19

Construct	Formalism
Clock-Driven Tasks	see External Event-Driven Tasks
Internal Event-Driven Tasks	...
External Event-Driven Tasks	...
Multitasking with Static Priorities	see Selection of Ready Queue with Static Priorities see Selection of Task with Static Priorities
Selection of Ready Queue with Static Priorities	...
Selection of Task with Static Priorities	...
If-Then	...

Figure 20

**Selection of Ready
Queue with
Static Priorities**

```

If      CPU <CPU name> available
Then   ready queue, priority <n>
        selected ready queue priority <n>

If      CPU <CPU name> available:
Then   ready queue, priority <n+1>
        selected ready queue, priority <n+1>

Unless ready queue, priority <n>

Repeat Unless until num_priorities = 1

Repeat If - Then (2) until num_priorities = 1
  
```

Figure 21

**Selection of Task
with
Static Priorities**

```

If      Task <task name>, priority <n> selected
Then   ready queue, priority <n>
        <OS>

If      <OS>
Then   Task <task name>, priority <n> START
        Task running
  
```

Event: <OS>.process scheduler

Time associated with event: <OS>.process scheduling <service time>

Figure 22

**Internal
Event-Driven
Tasks**

*If Event <event name>
 suspended queue, priority <n>
Then ready queue, priority <n>
Repeat Event <event name> until Event = null*

Figure 23

Results:

Performance Analysis

System throughputs (task and service throughputs)
System resource utilizations (hardware and software)
Average queue lengths of system resources
Occurrences of system deadlock
Livelock

Safety Analysis

Cost Analysis

Reliability, Maintainability, and Supportability Analysis

Fault Tolerance Analysis

Figure 24

INTEGRATION I

Strengthening the Systems/Software Engineering Interface for Real Time Systems

Mack Alford
Ascent Logic Corporation

ABSTRACT

As both Systems Engineering and Software Engineering mature, care must be taken to ensure that the interface between the two disciplines supports the passage of information as 'smoothly' as possible (i.e., is neither labor intensive nor error prone). Several current problems are identified, and a solution to the "Babel of Notations" problem is proposed.

1. INTRODUCTION

In his fantasy series about the mythical magical world of Xanth, Piers Anthony describes a chasm full of dragons which divides Xanth. The chasm has a "forget spell" attached to it, so that one forgets it is there unless within 50 yards of it. Characters in his novels are continually making plans, and then suddenly discovering (actually, re-discovering) the chasm as they near it, and having to battle dragons to get across. Of course, after they have finally crossed it, as they depart they forget that it exists.

This is a good analogy for the chasm that separates systems from software engineering -- a chasm with a "forget spell". Software engineering literature is full of discussions of how to do requirements and designs of real time software residing in embedded systems, but it is only when they start to work on a real project that they rediscover that they must get their requirements from the systems engineers. Similarly, systems engineers perform their front end studies, with full knowledge that software will have to be developed; but it is only WHEN they get ready to turn over requirements to the software engineers that they rediscover that WHAT they are ready to turn over does not meet the perceived needs of the software engineers.

During the past 30 years, the complexity of systems being implemented has by any measure increased by several orders of magnitude (e.g., size of code, size of memory, number of instructions per second). During that time, the interface between the systems and software engineers has suffered significantly. With the advent of automated tools for both the systems and software engineers, new problems have prevented the desired "seamless" transit between tool sets, and it is becoming imperative that this interface be smoothed out. Unless this problem is solved, the interface will remain both labor

intensive and error prone, and will thwart efforts to improve both productivity of the development process as well as the reliability of the end product operational system.

The purpose of this paper is to trace the development of this interface from its inception in the 1960s, identify the current issues, and to propose an approach to one of its thorniest problems -- the mapping of systems engineering notations onto software engineering notations.

2. BACKGROUND

The discipline of Systems Engineering gained prominence in the late 1950s, because it was viewed as a solution to the problems associated with the development of systems of high complexity with engineers from multiple disciplines. It was so successful that in the mid 1960s its use became mandated by the Department of Defense on all military systems, and all military contractors sponsored training courses for their front end engineers to gain a working knowledge of its concepts.

In 1968, the term "software engineering" was coined, and concepts of "programming" and "software development" were matured into those of "engineering the software". An attempt was made by systems engineers to treat software as "just another component". Systems engineers allocated functions to software components, and specifications were written to document those allocations; then software developers developed software to satisfy the requirements in the specifications.

To understand the issues related to the interface between systems and software engineering, it is useful to step back and review both the fundamental culture of systems engineering, and the way in which the interface was viewed as software engineering was developing. This then sets the stage for understanding the current interface between them.

2.1 Culture of Systems Engineering. The basic concept of systems engineering presented in the 1960s is rather simple and elegant, as illustrated in Exhibit 1. Systems engineers are responsible for translating customer goals, desires, and "requirements" into an integrated functional description of the black box behavior of a system and associated performance. This behavior is reviewed with the customer to gain concurrence, and then

these functions and their performance are decomposed and allocated to components, thus providing a systematic method of exploring the design space. Each design is evaluated by component developers for feasibility, cost effectiveness, schedule and risk, and the process iterated until an optimized (or at least acceptable) design is found. In addition, the designs are evaluated by the engineering specialties (e.g., reliability, availability, logistics, human interface, training, manufacturability) to ensure that these aspects of the design are acceptable as well. When there is consensus on feasibility, acceptability, and cost-effectiveness of a design by all players (including the customer), this design becomes the baseline description.

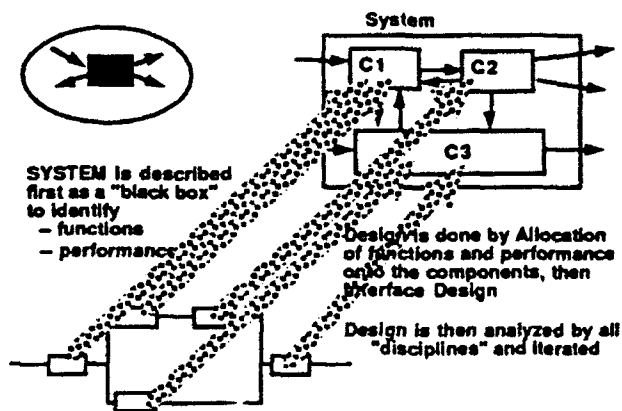


Exhibit 1 Fundamental Concepts of Systems Engineering

At this point, development shifts to the design of components to satisfy the allocated requirements. Systems Engineers monitor the design to ensure that the requirements will be satisfied, monitor the integration and test activities, and process the hundreds (or even thousands) of change orders resulting from changing customer requirements and/or component designer feedback.

The criteria dictating the level of detail of the functionality allocated to the components is simple: the black box behavior of the component is described to the level of detail necessary for the component developers to complete their design without reference to how other components are being developed. In particular, all interactions are to be identified and specified. It is the Systems Engineer's job to ensure that if all component developers satisfied their requirements, then the components will integrate correctly to satisfy the customer's requirements. This approach allows fairly complex systems to be developed, because component developers need only to satisfy their allocated requirements and interfaces -- this limits the amount of information needed to develop any component. The components are usually divided by discipline (e.g., propulsion, structures, electronics) so that one trained in that discipline can complete the designs of the components.

The component developers thus play three different roles:

- during the systems analysis and early design phases, the component developer provides estimates of feasibility, cost, schedule, and risk implications of proposed allocation of functions and performance to the component to aid the systems engineer in defining the system black box behavior. These responses are usually performed in working groups, with little formality, but are vital to the assessment of feasibility of a system design and to support tradeoffs.
- during the end of the system design phase, a system specification containing the allocation of requirements to a component is reviewed by the component developer to ensure agreement with proposed cost, schedule, and risk assessments; and
- during development phase, the component developer is responsible for development of the component to satisfy the requirements. When completed, the component developer assist integration and test.

The mechanism for defining the system behavior originally promulgated in the 1960s was to use the Functional Flow Block Diagrams (FFBDs). This provided a hierarchical approach for the definition of the timelines of function execution. The original applications of the approach was the design of a missile and its launch time line, so there was a significant bias towards representation of sequences and concurrencies of functions. Exhibit 2 presents an example of the notation.

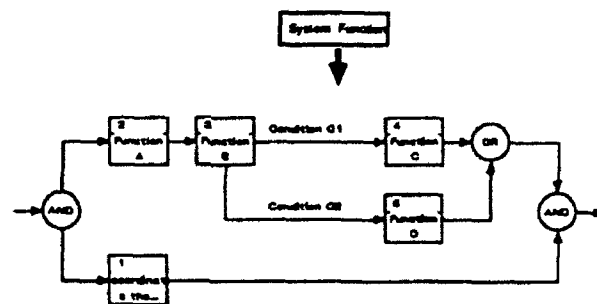


Exhibit 2. Sample FFBD - Hierarchy of Functions

The importance of the choice of FFBDs to the interface between Systems and Software Engineering will be discussed further below.

2.2 SE/SWE Interface in the 1960s. During the 1960s, selection of the computers for a system was driven by the environmental and sizing requirements. Reliability for the computer component was dictated by the reliability of the computer hardware. By today's standards, the computers were quite small in both memory and execution speed, and hence the complexity of the software was limited. Thus the allocation of functions to the

computer was equivalent to the allocation of functions to the software with sizing constraints.

To get an appreciation for how the interface between systems and software engineering has evolved, it is instructive to read the systems engineering textbooks of the 1960s and early 1970s. For example, in his book *The Management of Systems Engineering*, Wilton Chase devotes an entire chapter to "Computer and Software Systems Engineering". Some interesting excerpts:

"Designing a computer program ... requires the application of pure logic for devising the information processing routines. ... Because computer programming is strictly an "intellectual exercise, its effective system engineering is the most difficult aspect of designing a complex man-machine system." [Chase, p 93].

"The most critical step in a software design and development effort is the startup requirements analysis. The miss-allocation of software requirements occurs when the analysis and definition is split among several functional organizational areas making proper technical coordination nearly impossible. An effective means of avoiding this problem is to assign the total responsibility for determining software requirements to a central systems engineering activity." [CHASE, p.100].

Topics covered in this chapter include determination of computer capacity, definition of functions, computer/software design tradeoffs, software design (including flowcharting by the systems engineer), coding, documentation, human interface, and testing. Detailed design and coding, and unit test were relegated to "programmers". In other words, systems engineers were in charge of the top level "software design".

This interface was somewhat similar to the interface between systems engineers and analog control engineers: the analog control loops were developed by the control engineers, and then imposed on the component developers as requirements.

The conclusion drawn from this discussion is that during the 1960s the software was not viewed as a component on the par with other components -- to the extent that software was considered as a component, the systems engineers were in charge.

2.3 Trends in the late 60s and 70s. During the next decade, a number of trends occurred which made the interface between systems and software engineers more complex.

First, as the computing hardware became larger, the size and complexity of software became larger. This had several predictable consequences. The design of the software into units developed by programmers became increasingly important so that more software developers could be used, and hence management of multiple developers became a critical problem.

To cope with this, software development was upgraded into Software Engineering, with emphasis on design techniques, cost estimation, design for maintainability, etc. Thus software developers were no longer regarded as "technicians", but as a separate discipline, which should be in charge of the software component. Structured Analysis techniques, using some version of Data Flow Diagrams, began to be used to discipline the software design process.

In addition, software became a Configuration Item (i.e., a component), with a software developer placed in charge of it. This seemed to place software on the same par as other components, but there was an important difference -- the computing hardware was usually regarded as a separate component, selected by systems and hardware engineers to satisfy cost, reliability, capacity, and logistics requirements. Thus the primary issue facing the software manager was whether all of the software could be developed on time and fit inside of the pre-selected hardware. This meant that the software manager was not a full member of the component development team.

The interface that evolved between the systems and software engineers during this time had some unfortunate characteristics. Systems engineers still defined functions of the system (organized by the FFBs), and allocated some of the functions to the software component; and these allocations were documented in specification documents (e.g., MIL-STD 490 B5 Computer Software Configuration Item). Software developers then began their software analysis from these documents. Thus the information content developed at the system level (with timeline orientation) was reduced to textual testable requirements on functions and performance, which was then translated into a data flow diagram terminology. This incompatibility between the system and software descriptive was partially masked by the need for an intermediate hardcopy document; non the less, it is a problem that has remained into the 1990s.

Finally, as systems complexity increased, the systems engineers abrogated their responsibility to provide complete requirements to the software component, and the software developers did not pick it up. Although the concepts were available (e.g., [ALFO]), few software specifications provided a complete set of testable statements constraining the accuracy and response times of each required action of the software (i.e., input, condition, output) residing in its computer. The value for such requirements in such a format has not been recognized by the majority of practicing software engineers even today.

In addition, the systems engineers abrogated their responsibility to define the interface between the system operator and the software component (and thus verify that the human/computer interface standards required for training were satisfied). These issues were deferred not just until software requirements, but in many cases past preliminary design and even detailed design until the coding phase, where they were not reviewed at all. This became the

prime cause for the generally wretched human computer interfaces exhibited by the generated software.

2.4 Trends in the 80s. During the 1980s, the capacity of processors continued to increase, and hence increase in system complexity got worse. But some new trends emerged that exacerbated the systems/software engineering interface problem.

First, as processors got faster and smaller, and communications capability increased, most embedded computing systems became distributed. Unfortunately, the design of the distributed system became an orphan, claimed by neither the systems engineers, nor the computer hardware engineers, nor by the software engineers. The systems engineers did not claim it because it was perceived as a component design issue; the computer hardware engineers could not claim it because the critical issue was the allocation of processing to processors; and the software engineers were not prepared to deal with the issues of computer system reliability, logistics, etc.

Secondly, due to advances in computer chip technology, it became possible to make faster computer systems by use of specialized computer chips, thus requiring a hardware-software tradeoff. Again, such tradeoffs became an orphan -- the systems engineers couldn't do it because of the low level of detail needed to perform such tradeoffs, while the computer hardware and software engineers did not speak a common language needed to perform the tradeoffs.

Another consequence of the advances in computer hardware was that it became cost effective to place high performance engineering workstations on the desks of the engineers to provide aids for system and component specification and design. The electronic designers were aided by the CAE tools (e.g., schematic capture and simulation, the schematic layout tools). The software developers were aided by CASE tools. And finally, the systems engineers were provided with system design automation tools (e.g., interactive simulators, automated traceability support, and tools to support systems analysis and design synthesis).

As these tools matured, a consensus emerged that the requirements and design specifications should be executable to eliminate ambiguities, incompleteness and inconsistencies. In the CAE world, a standard executable model emerged (i.e., VHDL). Unfortunately, in the CASE world, a plethora of tools emerged using a variety of modeling notations (i.e., Data Flow Diagrams, Control Flow Diagrams, Petri Nets, SADT/IDEF0 diagrams, Object Oriented requirements models, the Mills Black Box notation, etc.). At the system level, some used various simulation models, while others used an FFBD notation extended into an executable timeline oriented behavior model. Yet the interface has remained the same as in the 1970s: systems engineers model the system, write paper specifications, then computer and software engineers

interpret the paper requirements to develop their own models to support the next layer of design. This interface is now even more labor intensive and error prone than previously. In many cases, the software engineering starts before the systems engineering has started, much less produce preliminary outputs to the software developers.

3. SUMMARY OF ISSUES AND SOME APPROACHES

Analysis of the above trends gives rise to six general issues that must be addressed in order to strengthen the systems/software engineering interface:

- 1) **Where does systems engineering stop and software engineering begin?** Two critical parts of this issue are:
 - 1a) **who does the human/computer interface?** On the one hand, systems engineering is ultimately responsible for ensuring that the HCI satisfies the required training standards. On the other hand, we have painfully learned that rapid prototyping is necessary to define such interfaces early in order to estimate the size of the required software.
 - 1b) **who does the distributed design?** On the one hand, a systems engineering job must be done on the distributed computer component to deal with issues of reliability, availability, logistics, sizing, etc.; yet much of the fault tolerance will be implemented in software.
- 2) **How detailed is the specification of the computer system/software?** It is a systems engineering responsibility to define the interactions between components to a sufficient detail that component developers can develop the components independently. This argues for an executable specification of the computer component, and equally argues for an executable specification of the software component at the (stimulus, condition, response) level, with performance requirements on response time and accuracy for specified arrival rates. Anything less than this is almost guaranteed to contain ambiguity and/or be incomplete.

This imposes requirements on both sets of participants -- the requirements for the systems engineer to generate such executable specifications, and the obligation

for the software engineer to demonstrate that they have been satisfied (i.e., the specified sequences of stimulus-condition-response behavior have been preserved). Although such specifications are currently feasible, the culture of systems engineering is only now beginning to recognize its obligation.

How is the specification information passed -- in a paper specification or by "database" on electronic media?

The CALS initiative is clearly moving towards electronic media for capturing relevant requirements/design information. One emerging view is that the paper specification should be a "text view" of the "model" defining component behavior, and that the data base of allocated behavior should constitute the "real" specification. Note that unless this is the case, the transition from systems engineering model to paper to software requirements model will be both time consuming and error prone. This gives rise to the next problem.

What is the notation of the model passed to software engineering?

The problem here is that systems engineers have their notations (e.g., time oriented FFBD notations, and their extensions), and software engineers have their notations (e.g., DFD or concurrent state machines). It does no good to insist that one of them change to using the notation of the other -- they have two different cultures, and cultural inertia and cost of training suggests that this solution is infeasible. On the other hand, if two different notations are used, then the mapping from one to the other must be automated, or the interface will be time consuming and error prone, not just for the initial specification, but for each change thereafter.

A significant problem here is that there is not a single software engineering notation -- Al Davis' book [DAVIS] describes a number of different notations in use today (e.g., DFD, CFD, Petri Nets, Concurrent State Machines, StateCharts, object oriented analyses, etc.). Definition of an automated mapping from system to software requirements is perhaps the biggest theoretical and practical problem relating to the interface. An approach to solving this problem is presented in the next section below.

What is the nature of the dialog (including the feedback) between system and software engineering? For example:

- when required processing is allocated to the computer component (particularly the response times and accuracy of processing), feedback is required on the feasibility and cost/schedule implications of such allocations (necessary to support h/w s/w tradeoffs);

- when processing requirements are allocated between computer and operators, a rapid prototype may be required to ensure that the software behaves as the user expects

Again, the issue here is to provide a rapid feedback mechanism to speed up the system design process. Finally,

- 6) If the systems engineers provide an executable specification of the software component viewed as a black box, how will the software engineering methods demonstrate (or prove) that the black box behavior (sequences of stimulus-condition-response) has been (provably!) preserved by the software design?

Current software development techniques are not currently oriented towards this problem.

4. REDUCING THE BABEL OF NOTATIONS

As noted above, one of the thorniest problems with the interface between systems and software engineering (with or without a CBSE role) is the problem of differing notations. The philosophy of Ascent Logic Corporation recognizes that, since systems engineering is the interdisciplinary engineering discipline, it is the obligation of the systems engineer to present information in the language of the component developers; hence it is the obligation of the systems engineering tools to provide an automated interface to downstream tools. The approach developed to address this problem is four fold:

- a) system behavior is described using "behavior diagrams (BDs)", which are an executable extension of the FFBDs. An element-relation-attribute-structures data store is used to keep information about these diagrams, their contents and traceability.
- b) as additional notations are considered, an attempt is made to perform a mapping from the BDs onto the notation -- the tool data store is extended to capture any additional information which is not yet represented in the current notation; and
- c) an editor is developed to generate the new notation from the data store using projection (i.e., finding the applicable subset of information needed to display the notation) and rules needed to display this information subset in the notation's syntax.
- d) information is output in the input syntax of the downstream tool which is used by the downstream developer. Hopefully, there is a standard for the syntax and semantics of such information which is accepted by many different tools (e.g., VHDL for CAE tools, CDIF for CASE tools).

The working hypothesis is that, since the BDs can be used to describe the observable behavior of any system or component viewed as a black box, then there should be a mapping onto other notations which are used to describe the same "black box". So far, this hypothesis has withstood the test of a number of other notations. After a presentation of the Behavior Diagram notation, an overview of the mappings to a representative set of notations is presented -- the mappings onto all known notations would surpass the page limitations of this paper.

4.1 Behavior Diagrams. The behavior diagram notation was developed by merging the concepts of the systems engineering Functional Flow Block Diagram (i.e.,

functions, sequences, selections, concurrencies, timelines), flow notations (i.e., flows of items between functions, as in IDEF0 or Data Flow Diagrams), Graph Models of Computation (i.e., showing multiple exits of a Function), Hierarchy of control concepts (i.e. defining replicated concurrent functions), and explicit labeling of function exit conditions and performance. The result was an executable notation which could be used to precisely define the intended behavior of a system. It was then augmented to describe interface designs and fault detection/recovery.

The foundation of the behavior diagram notation is the concept of DiscreteItems processed by DiscreteFunctions. A DiscreteItem may have contents, but arrives logically as a unit at an identifiable moment in time. The DiscreteItem can be used to represent either a physical thing (e.g., a peach) or a set of data. Some DiscreteItems (called state items) contain the partially processed results of previous functions operating on previous input items, and are passed down to subsequent functions for use in processing future arriving items. The DiscreteItems entering or exiting the system boundary are by definition the "observables" of the system. The DiscreteItems are represented on a graph by a shaded oval containing its name.

Exhibit 3 presents a DiscreteFunction, represented graphically by a shaded rectangle. Time flows from top to bottom, so the line at the top of the DiscreteFunction carries the enablement from a previous function. When enabled, the Discrete Function waits for the arrival of a Discrete Item -- in the diagram below, only one DiscreteItem A is shown, but more than one is possible. When the first item arrives, any combination of the output items (e.g., B and/or C) can be generated (including the state item S2), and one of its exits is taken (which enables some subsequent function). after a finite duration. If a non-designated item arrives, this is assumed to be an error (e.g., input out of sequence). The exits are labeled with the names of the conditions (e.g., C1 and C2) to be met to take the exit. Exits can be classified as normal, exceptional, or "timeout" (which is taken if no input arrives within a designated period of time). The DiscreteFunction can also be defined to require a designated resource amount, and if this is not available when the input item arrives, the function will wait for the resource availability.

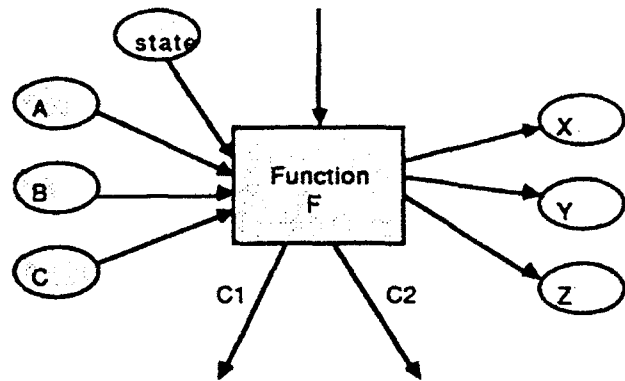


Exhibit 3 Example Discrete Function

A DiscreteFunction can be decomposed into a "Response Net", or RNet, to define how the contents of the input items are used to determine conditions under which each output item and condition is selected, and are used to generate the contents of the output items. The definition of a DiscreteFunction is a mild extension of the "function" of a finite state machine -- it is allowed to generate combinations of outputs (not just a single output), its execution may require the availability of a designated resource, and the state items (e.g., containing state information) are explicitly defined. The RNet and their contents are equivalent to a completed decision table defining the stimulus-condition-response of a function. An example of an RNet is shown below, which accepts A, B, or C and outputs either X and Y or Z or nothing (depending on the arrival and value of a condition CC1, then generates the state item and selects the appropriate exit.

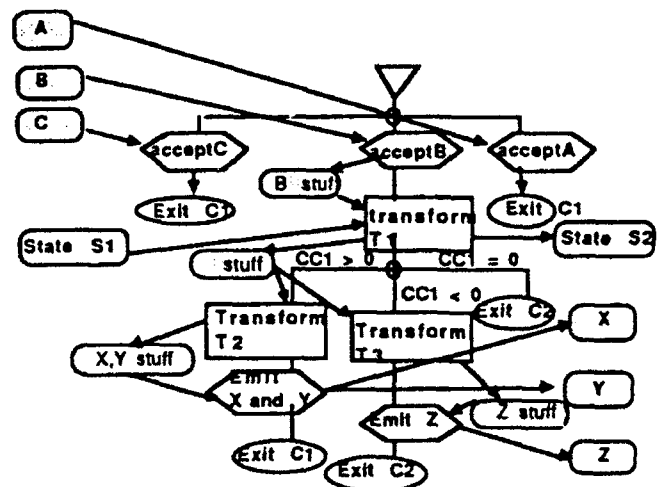


Exhibit 4 Example Response Net

Sequences of Functions are described with a Function Net, or FNet containing functions, as shown in Exhibit 5. Time flows from top to bottom (indicating arrival of items to be processed) and left to right (reflecting inputs transformed into outputs). Looking at the black box

boundary of Exhibit 5, note that first a Peach arrives, then a Can; that first a peach Skin, then a Pit, and then Canned Slices exits the boundary -- these are all explicitly observable. The processing is described as a sequence of three functions. Note that state items pass from "Skin a Peach" to "Slice a Peach" and from "Slice a Peach" to "Can a Peach", and since they are "inside the box" they are not observable. The Function "Can a Peach" cannot execute until the "Can" arrives.

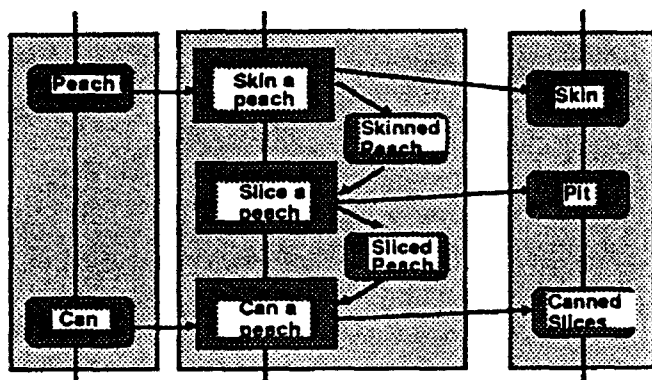


Exhibit 5 Example Sequence

This notation is executable, i.e., it can be executed to yield the times of the outputs from the times of the inputs and the duration's of the functions. The Figure below presents a sample timeline. Note that the empty bar indicates a period of time when a function is enabled but waiting for an input, and a dark filled bar indicates the duration required for the execution of the function. Outputs are available when the function completes.

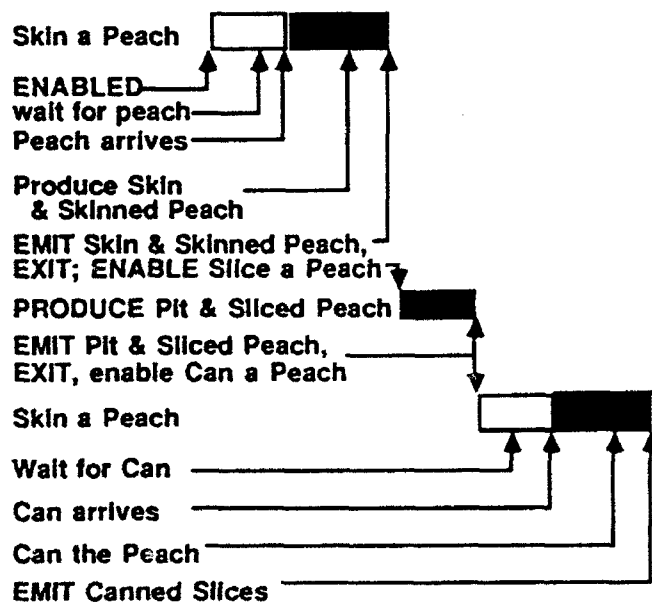


Exhibit 6 -- Sample Timeline

The functions can be placed into graphs containing not just sequences, but selections, iterations, loops, concurrencies, and replications as well. The notation for these graphic constructs appears below.

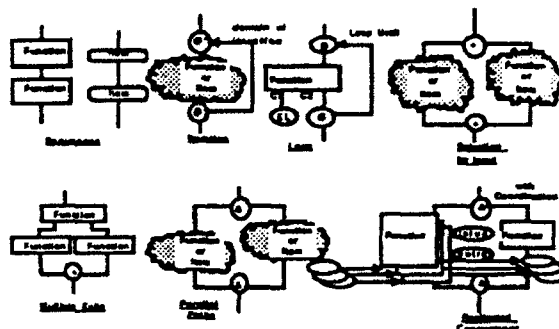


Exhibit 7 Behavior Graph Constructs.

The replication construct requires a special word of explanation. One defines the domain of replication (e.g., for each aircraft in track, for each user of the system), and then defines one function per replicate. In this way, users do not have to deal explicitly with "indices of functions" in order to define them. Finally, a coordination function is defined, which accepts status from the replicates and generates controls back to them; the coordination function is responsible for detecting and resolving all conflicts between the replicates (e.g., two aircraft collide, or giving priority to certain users if there are insufficient resources to service all).

To deal with large models, a graph of functions can be aggregated into a "TimeFunctions", i.e., a function which inputs and outputs specified sequences of items. Functional Decomposition then reverses the aggregation process, defining a graph of functions which preserve specific properties of the original function (i.e., input/output content and sequence, number and kinds of exits, and ability to calculate the performance of the parent function from the performance indices of the functions on the graph. The aggregation or decomposition procedure can be recursive to organize a graph of arbitrary size and complexity into a hierarchy of functions and their decompositions to support understandability.

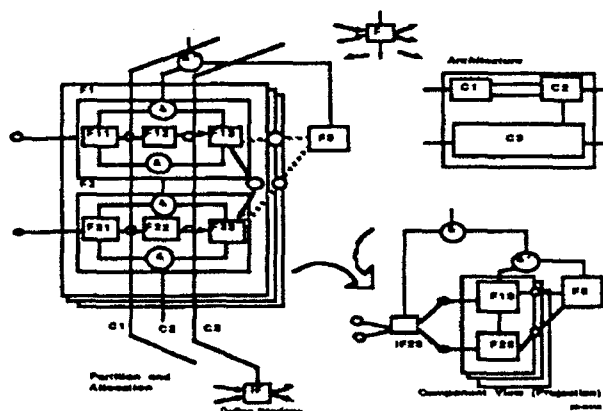


Exhibit 8 Allocation of Functions to Components

When the desired behavior of a system is defined, functions are allocated to the system components. The Figure below illustrates this process. The black box behavior of the system is decomposed to the level that functions can be partitioned and allocated to the components of a postulated architecture (e.g., C1, C2 and C3, shown in the upper right). Note that the allocation yields the requirements for a new interface function labeled IF, which is decomposed and allocated between sender and receiver (and possibly a communications component). This process can be recursively applied to yield layers of interface design. The resulting allocated functions, including those implementing the interface design, then are extracted to yield the black box behavior of a component. The extraction process can be implemented by projection operators.

The RDD notation thus provides the system designer with the ability to define an executable description of the desired behavior, and its allocation to components. It supports separation of concerns (e.g., separation of normal from exceptional behavior, single object behavior from behavior to coordinate concurrent functionality, normal from interface behavior).

4.2 Systems Engineering Notations (FFBDs, N-Squared charts, IDEF0). The BD notation is mapped onto the Systems Engineering notations in order to present practicing systems engineers information in a notation in which they were trained. Although the BD notation was synthesized from the FFBDs and other notations, it is useful to actually construct publishable FFBDs from the BDs. For example, the FFBD in Exhibit 2 was created from a BD automatically by eliminating the inputs, outputs, and conditions, and displaying the resulting function sequences and concurrencies in a left to right format.

A corresponding N-Squared chart is constructed by eliminating the function sequence, concurrency and conditions, and external inputs and outputs. The resulting functions and their flows are represented by placing the functions on the diagonal of a matrix, then placing a circle denoting flow on the (i,j) matrix element to represent the flow from function i to function j. A publishable N-Squared chart is presented in Exhibit 9.

Many systems engineers from the manufacturing area use the IDEF0 flow notation (a variant of the SADT notation developed by Doug Ross, see [ROSS]). This notation describes essentially the same information as the N-Squared chart, but describes external inputs and outputs as well as internal. The functions are arranged on a diagonal as with the N-Squared Chart, but labeled arrows are used to describe flows. In addition, if a flow is designated as "data", it enters the side of a function box; if the flow is designated as "primarily a control", then it is shown to enter the top of a function box. These are generated from BDs by ignoring the sequence-conditions-

iteration-concurrency-replication information, and defaulting the flows to the "data" mode, and allowing the user to later designate it as a "control flow". Exhibit 10 presents the IDEF0 diagram that results from the application of this procedure.

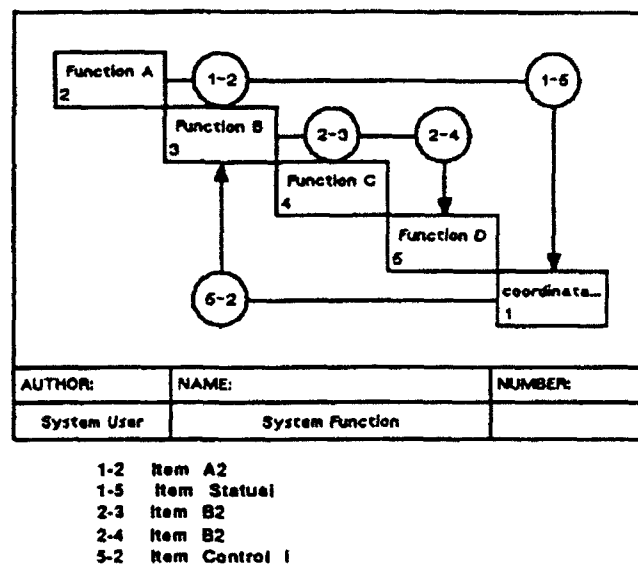


Exhibit 9 Example N-Squared Chart

These transformations are performed automatically by the RDD-100 System Designer using an interactive editor for each type of diagram. When the flows are modified in one diagram, the flows in all other diagrams are updated when selected, so that the different views of the information in the data store cannot be "out of synchronization".

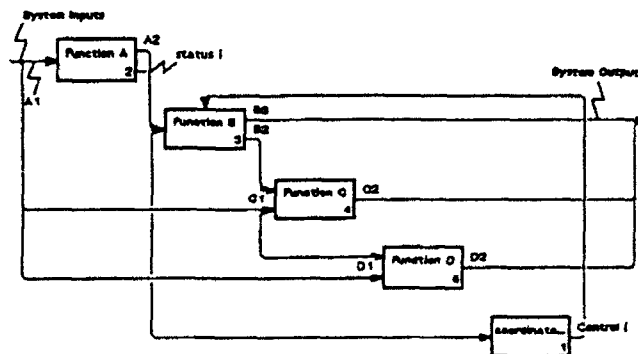


Exhibit 10 Example IDEF0 Diagram

4.3 Mapping to Petri Nets. I consider Petri Nets to be the assembly language of behavior. Simple Petri Nets are used to describe sequence, selections, and concurrency as shown below:

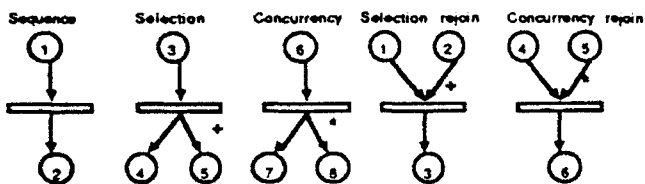


Exhibit 11 Petri Net Constructs

Sequences is represented as follows: a "token" is placed into the Place 1; at some point, a transition will "fire", resulting in removing the token from place 1 and inserting a token into Place 2. Selection is indicated by the transition removing a token from place 3 and inserting tokens into either place 4 or place 5. Concurrency is indicated by removing a token from place 6, and inserting a token into both place 7 and place 8. Thus the "+" indicates a selection, while the "*" indicates that all exiting places will be filled. Rejoins of selections and concurrency are indicated as shown above.

For a selection rejoin (indicated by a "+" on the upper side of a transition), if there is a token in either place 1 or place 2, when the transition "fires", the token is removed and a token is inserted into Place 3. For a concurrency rejoin (which is actually a synchronization point), when there are tokens in both places 4 and 5, a transition can fire which removes both tokens and inserts a token into place 6.

Colored Hierarchical Timed Petri Nets

Colored Hierarchical Timed Petri Nets are an extension of Petri Nets in which:

- the tokens can be specified to be of a specific type, and to contain a specific subset of "data" (e.g., an index);
- the "places" are "bags", i.e., can contain many different tokens
- the transitions can be specified to take into account the token types and value-- this allows one to have many different tokens on a graph to represent processing of multiple objects, to specify transitions as occurring only with tokens with the same index, and to specify the transition in terms of their transformations for mapping the input tokens into the values of the output tokens.
- a fragment of a Petri Net can be aggregated into a "place" which preserves the input and output arcs of the fragment. This allows a large problem to be described using a hierarchy of Petri Nets.
- a transition can be specified to take a specific amount of time, and can also be used to specify a "timeout".

The mapping of the BD notation onto Colored Petri Nets is fairly straightforward, and can be done in two parts. First, the DiscreteFunctions are mapped onto a Petri Net Fragment; then the BD constructs of sequence, selection, concurrency, iteration and replication are mapped onto Petri Net fragments.

Consider the Discrete Function shown in Exhibit 3. One can describe this DiscreteFunction as having four phases -- the enablement phase (i.e., arrival of function enablement and state item); the triggering phase, when one discrete items A, B, or C arrives; the calculation phase in which some combination of the outputs X, Y, and Z are generated; and finally the exit phase, when one of C1 or C2 is taken (this example ignores the resource, which could be shown as an additional branch).

This DiscreteFunction F can be represented by the Petri Net shown below. The enablement phase is modeled by the arrival of tokens to the ENTRY, and the token carrying the state information. -- a token would thus be placed at Place 2. A colored token could be used to represent the arrival of DiscreteItems A, B, or C, and when the transition fired, its content would be inserted into Place 1, and thus the transition after 1 and 2 could fire,

resulting in a token being inserted into place 3 (and all those to the right of it). The three concurrent Petri Net branches model the generation of the outputs (only one of these is numbered for simplicity). The transition after Place 3 determines whether an output should be generated, and places a token in either Place 4 (no output is to be generated) or Place 5 (the output X is to be generated). The transition after Place 5 generates the output token X, and places a token in Place 6; in either case, a token is now placed at 7. When all of the branches for the calculation of X, Y, and Z are completed, then the transition after Place 7 determines whether a token is to be inserted into Place 8 or Place 9, resulting in the output of a token corresponding to either condition C1 or C2.

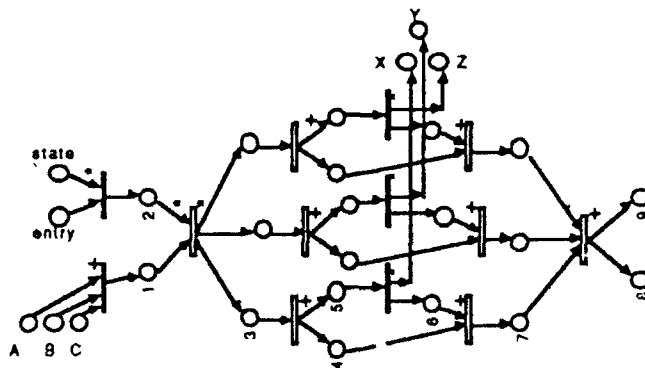


Exhibit 12 Example Petri Net of a Discrete Function

Thus a DiscreteFunction can always be exactly modeled by a Petri Net with input places corresponding to each of its input Discrete and State Items and its input arc, output places corresponding to the output Discrete and StateItems and exit arcs. In a similar fashion, all of the major graphic constructs can be implemented using Petri Nets. The result is that any Behavior Diagram can be mechanically translated into a single large Petri Net, or onto a Hierarchical Petri Net. A consequence of this

mapping is that many of the proofs about Hierarchical Colored Timed Petri Nets can be applied to Behavior Diagrams.

The reverse mapping is much more difficult, because the user may not have imposed the same sort of regular structure on the Petri Net as do the DiscreteFunctions of the Behavior Diagrams. This is analogous to the difference between structured code in a higher order language mapped onto assembly language — the higher order structuring is projected out when the mapping occurs, and it is quite difficult to recreate the higher order structuring from only the assembly code.

4.4 Mapping to Data Flow Diagram Notations.

A Data Flow Diagram was originally developed to identify requirements for non-real time software systems. The notation is similar to IDEF0 and N-Squared charts in that it displays the flows between functions. However, it contains an additional construct — the "Data Store", defined to contain "state information". The Data Store is defined in Data Dictionaries as "containing" a list of flows (i.e., those input to and output from it). Thus the mapping of BDs to DFDs requires the identification of Data Stores, and the establishment of a "contained by" relationship between state items and a Data Store to which it is assigned. Note that this mapping is not unique, but can be made to be complete (i.e., every state item is contained in some Data Store). With these definitions, a Data Flow Diagram can be constructed from a BD by:

- defining a DFD process for each BD Function
- assigning state items to Data Stores
- representing the input/output relationships as labeled arrows between the processes and/or Data Stores

A set of "Control Flows" between the processes can be obtained from the BD graph by representing each enablement between functions as a Control Flow between processes. An example of such a mappings are presented below.

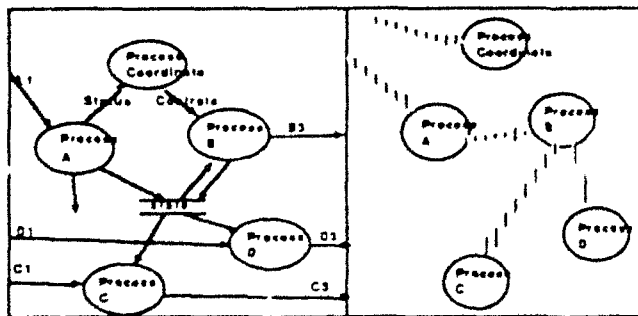


Exhibit 12 Example Data and Control Flow Diagrams

The BD information not represented on these diagrams includes the identification of the "observable" items input to and output from the system boundary (although this might be represented in textual descriptions of the flows).

the expected sequences of inputs, functions, and outputs, the replication of functions, and the conditions under which the control flows are generated. The mappings of the conditions onto the control flows is imprecise because of the replications. Hence the reverse mappings onto BDs requires the addition of the replication, conditions, and sequencing information projected out on the forward mapping. No automated mapping seems possible because of the missing information.

If executable descriptions are added at the bottom level (e.g., with Petri Nets as is done in ADAS), then the data flow model is transformed into a set of interacting concurrent state machines), described next.

4.5 Mapping to Interacting Concurrent Finite State Machines(CFSM).

There are a large class of models which represent a system as a collection of Interacting Concurrent Finite State Machines (e.g., VHDL, DFD/ADAS, CCITT SDL, object models). The key to mapping a BD model onto this style of model lies in the fact that, when every TimeFunction is replaced by its decomposition Behavior Diagram, the result is a very large graph containing a number of concurrent branches. Every concurrent branch is called an RDDProcess which satisfies the definition of a Terminating Finite State Machine. Every RDDProcess is enabled by some other process, receives items from and sends items to other RDDProcesses.

To turn this graph into a collection of Concurrent FSMs requires that each RDDProcess be mapped onto an FSM "component". This yields one FSM for each replicated RDDProcess. The interface design is then constructed to represent the passing of enablements along the BD Graph of RDDProcesses as the passing of messages between the resulting FSMs. This means that each RDD Process, when terminating in the start of a concurrency, would "send an enablement message" to the indicated FSMs; the FSMs would be augmented with a function to accept the enablement message to get started. An example of this process is presented in the figure below.

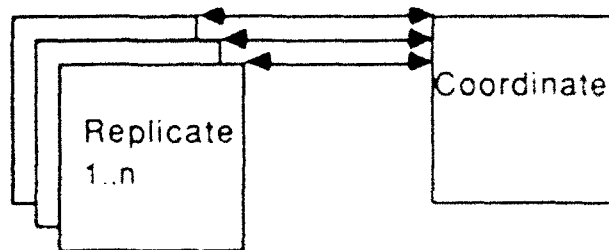


Exhibit 13 Example Concurrent Processes

This mapping has been prototyped for VHDL. Each arriving item is modeled in VHDL as a "signal", and the invocation of each DiscreteFunction in an RDDProcess is equivalent to the "call" of a procedure in VHDL. Thus the RDDProcesses map to VHDL processes, and the sequence

of functions in RDD maps to a sequence of functions in VHDL.

The information lost by this mapping is the hierarchy itself (which provides the reader with the ability to understand the intended sequences of actions), the expected sequences of inputs, and the identification of the expected sequences of normal processing.

The loss of the sequencing information turns out to be crucial. Even though the sequences of externally observable inputs and outputs can be recreated by hitting the collection of FSMs with a number of "test vectors", this process is extremely time consuming. Moreover, if each of only 20 FSMs has but 10 states, then the dimensionality of the states of the combined state machines is on the order to $10^{**}20$ -- far to large to systematically explore in a limited amount of time (and the dimensionality of 200 FSMs is on the order of $10^{**}200$). In its original hierarchy form, most of these states are observed to be impossible (i.e., a state of an FSM is not meaningful until it is enabled).

Thus the reverse mapping (from CFSMs to BDs) requires the effort of recreating the expected sequences of inputs and functions, and no automated technique is known to exist.

4.6 Mapping to a Single Extended State Machine (SREM, Mills box notation). Both the SREM and the Mills box notations are variants of descriptions of a single state machine representation of a system. In both cases, the system is described as having a single state, and the arrival of a input item yields the generation of an output item and a transition to some next state. The state machine is extended to allow an output "event" to become a subsequent input item. It is expected that an input will be completely processed before the next input is accepted.

Mapping the BD onto a single state machine requires several transformations. First, the set of concurrent state machines is obtained by recursively substituting BDs for functions and hence eliminating the hierarchy constructs. Next, the concurrent state machines are "serialized" by the addition of an external serializing function which does not allow the next input to arrive until the previous input has been completely processed.

The multiple state machines (and their interactions) could be viewed using various object notations. Note that each concurrent branch of behavior has given rise to a state machine which encapsulates the state information passed down the branch of processing.

Each of the multiple state machines can be collapsed from a Moore model into a Mealy model by "stateizing" the location of the token indicating which function is active. This is equivalent to adding a variable with enumerated values (i.e., the names of the active functions), then adding an initial function which accepts the input, determines the current state, then activates the appropriate function with appropriate state information.

All of the multiple state machines can then be collapsed into a single state machine with partitioned states (i.e., one partition per state machine). If many replicates are collapsed into a single state machine, then the interface function must determine which replicate is appropriate to the input message, and invoke the appropriate function. This is equivalent to the SREM notation. Similarly, a "box" notation of Mills can be similarly obtained.

The reverse notations share the same problems as the concurrent state machine notations. One must hit the single state machine with a number of test vectors to "unfold" the processing into the intended sequences of functions, and then in addition identify the allowed concurrency of operations.

4.7 Conclusion -- Automated Mappings Are Possible, and Mandatory. The primary conclusion drawn from the above is that it is possible to provide automated support for the mappings from the RDD systems engineering notation into the various software engineering notations. Much of the transformation is automated, but some require the addition of notation peculiar information (e.g., identification of "control flows" in IDEF0, identification of "data stores" for DFDs), but some faults can be automatically supplied (e.g., all flows are "data flows" for IDEF0, all state items contained in an RDDProcess are assigned to a default "data store"). The availability of such mappings strongly suggests that the use of RDD at systems engineering time provides a solution to the "Babel of Notations" problem, i.e., any of these notations can be obtained by automated means.

The reverse mappings appear to be much more difficult, requiring the addition of significant amounts of information not contained within the software engineering notations. In fact, the reverse mappings for the Petri Nets and the concurrent finite state machines require the re-creation of the structuring information -- a notoriously difficult task.

5.0 DISCUSSION

Although a smooth interface is required from Systems to Software engineering, the way in which the two disciplines have grown in the past 25 years has not satisfied this requirement. A number of problems with the interface have been identified, and some approaches for solutions have been discussed.

The mapping of the RDD Systems Engineering notation onto several software engineering notations suggest a solution to the "Babel of notations" problem. It also provides some valuable insights into the benefits and drawbacks of these software engineering notations (e.g., the lack of hierarchy, the lack of support for a separation of concerns of normal from exceptional behavior, the lack of representation of desired sequences of functions may provide some explanations for why the specification of

required software behavior has been so notoriously difficult).

Several open issues have also been identified: the necessity for a CBSE discipline to perform all of the functions of distributed design, and the necessity for software development methods to demonstrate preservation of allocated executable black box level behavior. It is suggested that software design methods approaches which describe constructive steps for allocating required processing onto units of code (and simultaneously prove that the behavior has been preserved) may provide a possible solution to this problem.

REFERENCES

- [ALFO] Mack Alford, "A Requirements Engineering Methodology for Real Time Systems", *IEEE Transactions on Software Engineering*, Volume 1, Number 1, 1977
- [ANTHONY] Piers Anthony, "Castle Roogna"
- [CHASE] Wilton Chase, "The Management of Systems Engineering", Prentice Hall, 1975
- [DAVIS] Alan M. Davis, "Software Requirements Analysis & Specification", Prentice Hall, 1990
- [LAVI] Jonah Lavi, "Overview of CBSE", *IEEE Computer*, Nov. 91
- [PETERSON] James Peterson, "Petri Net Theory and the Modeling of Systems", Prentice-Hall, 1981

START/ES — An Expert System Tool for System Performance and Reliability Analysis

Eric W. Brehm, Robert T. Goettge

Advanced System Technologies, Inc., 12200 E. Briarwood Ave., Suite 260, Englewood, CO 80112

Fred McCaleb

NASA Goddard Space Flight Center, Code 522, Greenbelt, MD 20771

Abstract

This paper describes START/ES, an expert system based tool for performance and reliability analysis of complex computer systems. START/ES provides an iconic system design capture interface, which allows direct manipulation of system design attributes and facilitates the exploration of a wide range of design alternatives. System design descriptions are automatically translated into mathematical models, which evaluate candidate designs in terms of both performance and reliability. Evaluation results can be analyzed by the automated reasoning component of the tool, which uses a rule-based approach to diagnose performance and reliability problems, and to recommend design changes for achieving system designs which are compliant with all requirements. Finally, the rule base is extensible and can be modified using START/ES's rule builder interface.

1. Introduction

Performance and reliability are key aspects of system effectiveness which must be considered during the design of mission-critical computing systems. Automated tools are needed to address the complexity of design alternatives, and to provide quantitative evaluation of system performance and reliability characteristics. The complexity of the interactions between design attributes is such that automated assistance is helpful in interpreting evaluation results, and in exploring the implications of alternative design strategies.

START/ES (for System Timing And Reliability Tool - Expert System) is an expert system based automated tool that addresses design verification of mission-critical computer systems. This paper summarizes the tool's capabilities for describing, evaluating, and analyzing systems.

The paper is organized as follows. Section 2 provides an overview of the primary components of START/ES. Section 3 summarizes the elements of START/ES system design representations. Section 4 describes the system design evaluation results produced by START/ES performance and reliability models. Section 5 discusses the START/ES expert system component, and provides examples of the types of design assistance produced at various stages of the automated reasoning process. Section 6 describes the START/ES rule builder interface which allows experts to modify the automated reasoning capability.

2. START/ES Overview

Figure 1 shows the relationship between the major components of START/ES. A number of different interfaces are provided to allow interaction with the tool by both system designers and by performance and reliability experts.

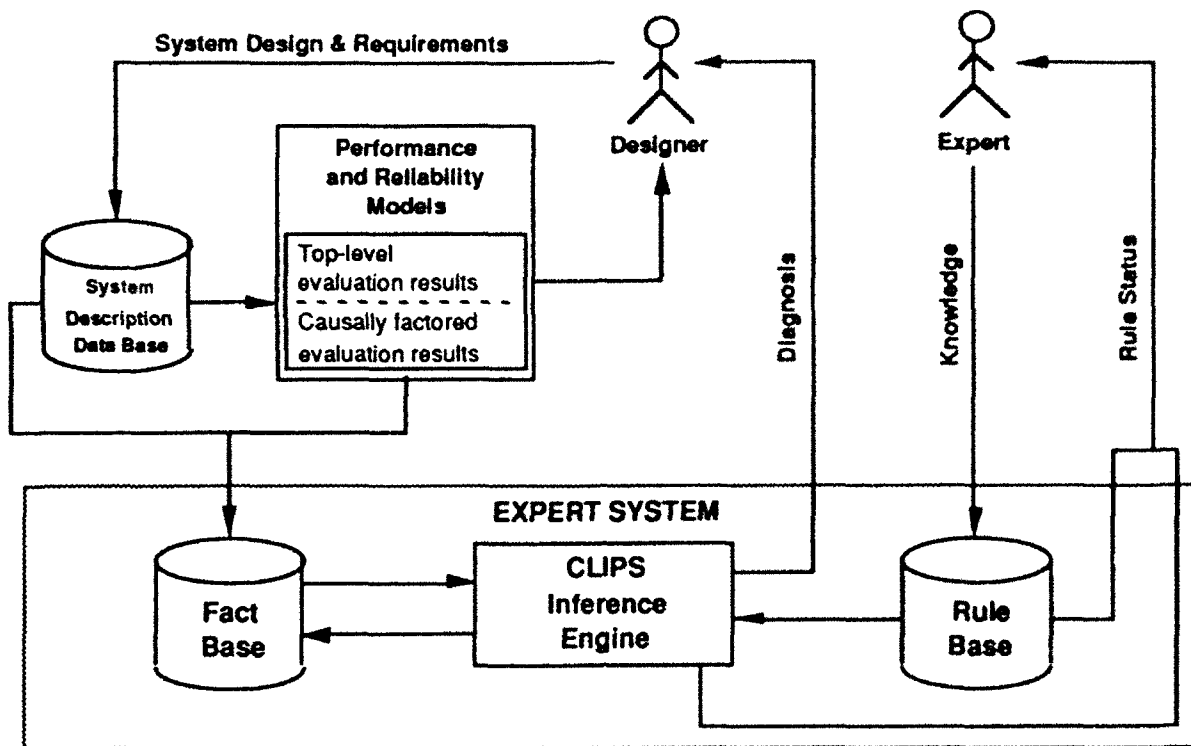


Figure 1: START/ES Components

The *system description data base* contains information related to the attributes and organization of hardware, software, and functional elements of a candidate system design. This information is entered using the graphical design capture interface originally developed for the START tool for integrated performance and reliability analysis [1].

The *performance and reliability models* provide analytic computational capabilities which transform system descriptions into quantitative indicators of system reliability and performance. These computational capabilities were also originally developed for the START tool. Top-level results produced by the models are displayed to the user as annotations on system description drawings.

The *expert system* component provides design assistance using expert system technology provided by the "C" Language Production System (CLIPS) shell [2] embedded within START/ES. The *fact base* containing system description and evaluation results data is queried during execution of the CLIPS *inference engine*, which exercises the automated reasoning process encoded in the *rule base*.

The expert system rule base is partitioned into three rule sets, each concerned with providing a particular type of design assistance:

- *Compliance assessment* rules determine whether a candidate system design meets its requirements. These rules compare top-level performance and reliability results against the corresponding requirement values specified in the system description.
- *Problem identification* rules identify the sources or underlying causes of performance and reliability non-compliance. These rules examine "causally factored" evaluation results, which indicate the contributions to delay and failure likelihood associated with specific system design

elements. Elements which are found to be significant contributors to non-compliance are asserted to be performance and/or reliability problems.

- *Design change recommendation* rules identify system design changes which will correct identified performance and reliability problems, and thus move a system in the direction of compliance with all of its requirements. The user can investigate the quantitative impact of recommended design changes by re-specifying appropriate system design constructs or parameters, and then re-executing the models.

Modification and extension of the rule base is accomplished using the "rule builder" interface. This interface allows rules to be expressed in an "English-like" manner more natural to an expert than the pattern matching syntax used in CLIPS. Various rule base maintenance utilities are also provided to assist the expert.

3. System Design Description

START/ES responds to the need for improved accessibility to performance and reliability modeling techniques, including consideration of the interaction effects between performance and reliability [3]. Intended users include system designers and analysts requiring performance and reliability design verification of mission-critical computer systems. These users are concerned with exploration of a broad architectural design space and verification of design alternatives. For this intended usage the *architectural variant* [4] level of design description is used in START/ES to expedite the analysis of performance and reliability.

The system description component of START/ES captures elements of a system design relevant to evaluation of performance and reliability characteristics. A graphical interface allows the user to specify the attributes and organization of the functional, software, and hardware components of the system.

System Functionality and Requirements

The basic functionality of a system is represented in control flow diagrams called *stimulus control flows* (SCFs); these indicate sequences of *primitive functions* which are triggered by arrival of an internal or external stimulus, and which typically result in one or more corresponding outputs. Primitive functions are connected by directional *flow connectors* which indicate the amount of data that is exchanged between functions. When multiple flow connectors emanate from a single function, *path probabilities* are assigned to these connectors to indicate the relative likelihood that each path is taken by an arriving stimulus.

System performance and reliability are measured relative to sequences of primitive functions delineated within SCFs by *stimulus-response markers*. The serial set of processing activities within the scope of a particular stimulus-response marker is referred to as a *response thread*. Figure 2 illustrates the graphical user interface that is used to define SCFs and stimulus-response markers.

System performance and reliability requirements are specified for response threads and for system resources (hardware devices and tasks). A *response time requirement* and an *availability requirement* are specified for each response thread. *Utilization limits* are specified for each hardware device and task in the system description.

Software and Hardware Components

Each primitive function within an SCF is attached to a *software module* which indicates the types and quantities of computing services needed to implement the function. Software modules are grouped into dispatchable units of software called *tasks*. The software functionality of the system is mapped onto

hardware by allocating tasks to specific hardware devices appearing in a *hardware architecture diagram*.

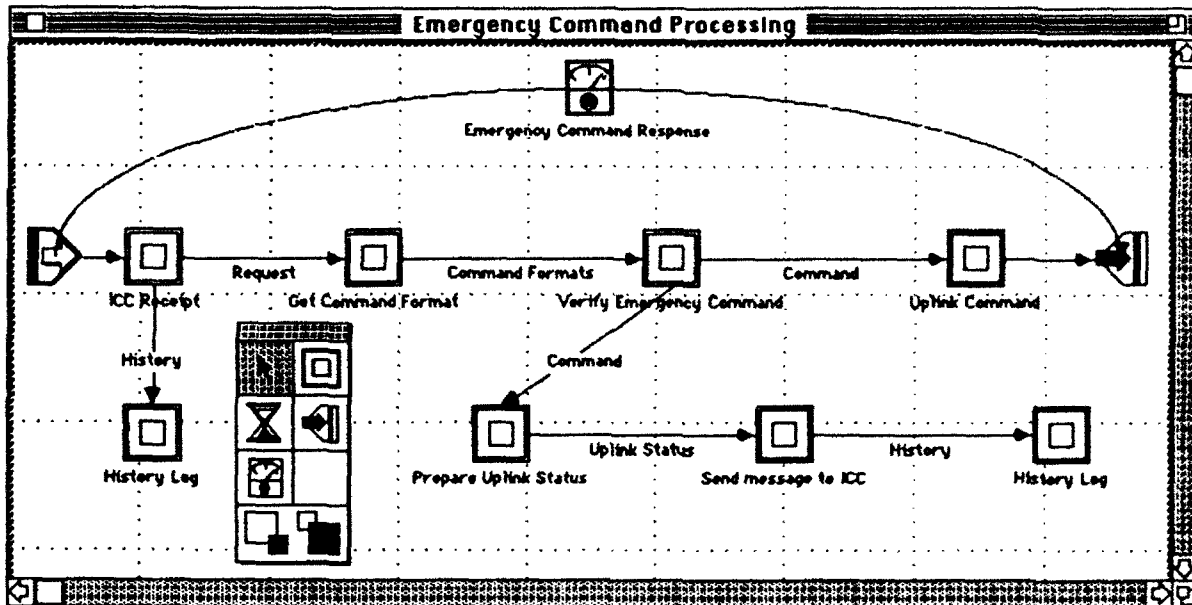


Figure 2: Stimulus Control Flow

The hardware architecture diagram identifies all of the *processors*, *communication devices*, and *storage devices* used in the system, and indicates how these are interconnected. Figure 3 illustrates the graphical interface that is used to define hardware architectures.

Table 1 lists basic performance and reliability parameters that are specified for each hardware device in the hardware architecture diagram.

In addition to device processing rates, system performance is affected by system service overhead rates for intertask communication, intercomputer communication, and data access; these are represented as attributes of *operating systems* specified for each processor, *communication protocols* specified for each communication device, and *processor overheads* specified for each storage device.

The hardware redundancy parameters specified for each device define a *hardware subsystem* consisting of a set of identical units which are designated either *active* or *backup*. It is assumed that the total processing load on the subsystem is shared by all active units, and that backup units are held in reserve to replace active units that fail. In each subsystem, *detection latency* represents the mean time to detect the failure of a unit, while *recovery time* is the mean time to bring an inactive unit into the active configuration.

The *transient error rate* specified for each device indicates the rate at which faults occur during use of the device which cause incorrect outputs to be produced, but which do not require physical repair actions to be performed. Subsystem error recovery options include *temporal repetition*, in which operations are repeated "temporally" on each active unit until two matching outputs are produced, and *N-modular redundancy (NMR)*, in which operations are replicated in parallel on each active unit, and a voting mechanism is used to detect and mask erroneous outputs.

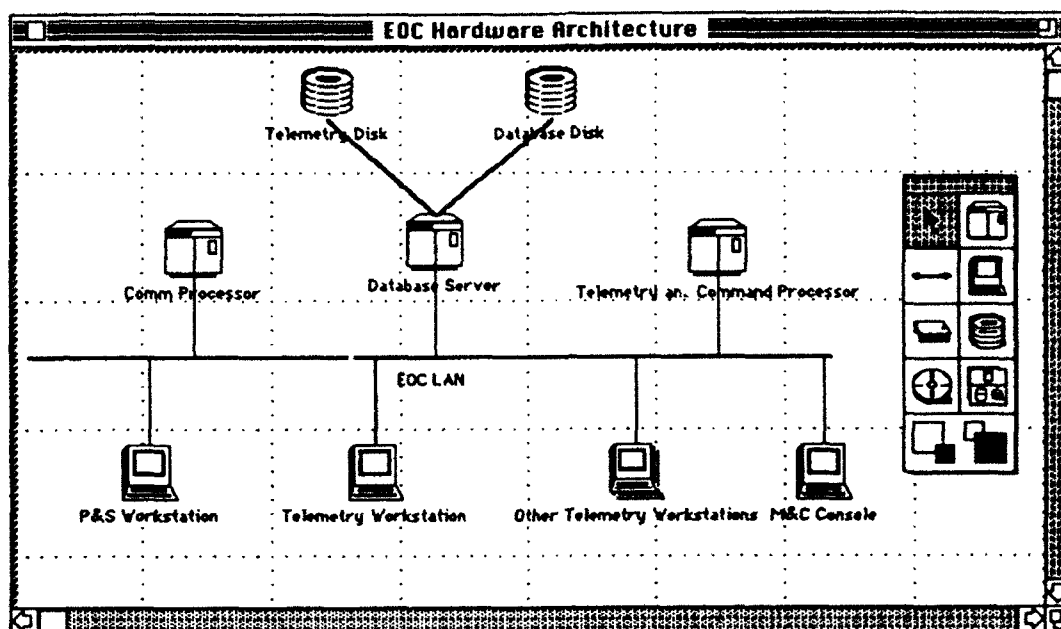


Figure 3: Hardware Architecture Diagram

Table 1: Performance and Reliability Parameters

Processing rate	Processors - Speed (in MIPS), multiplicity Communication Devices - Transfer rate, multiplicity Storage Devices - Transfer rate, access latency, multiplicity
Reliability	Mean time to failure (MTTF), transient error rate
Maintainability	Mean time to restore (MTTR)
Hardware redundancy	Number of active units, number of backup units Detection latency, recovery time
Error detection and correction	Temporal repetition (Yes/No) N-modular redundancy (Yes/No)

Hardware subsystems can be used to represent many different configurations, each of which provides some degree of standby redundancy, extra processing capacity, or error detection and correction capability. Table 2 indicates parameters that would be specified for some commonly used configurations.

START/ES also provides a means to account for the unreliability of software. A *mean executions between failure (MEBF)* parameter may be specified for each software module, indicating the mean number of invocations of that module between functional failures caused by a software design fault [5].

Table 2: Hardware Subsystem Configurations

Configuration	Hardware Subsystem Specification		
Simplex system (no redundancy)	Number active = 1	Number backup = 0	
Duplex system	Number active = 2	Number backup = 0	NMR selected
Triple modular redundant system	Number active = 3	Number backup = 0	NMR selected
Hybrid redundant system	Number active ≥ 2	Number backup ≥ 1	NMR selected
Loadsharing system	Number active ≥ 2	Number backup = 0	
Standby redundant system	Number active ≥ 1	Number backup ≥ 1	
Master/slave system	Number active = 1	Number backup = 1	

4. System Design Evaluation

START/ES automatically translates system design descriptions into mathematical models, then executes these models to obtain measures of system performance and reliability. Behavioral interrelationships between performance and reliability -- such as service degradation due to permanent hardware failure ("performability") and utilization of processing resources by error recovery mechanisms -- are accounted for through parametric interchange between the models of each type. The primary performance and reliability measures that are computed are as follows:

- *Thread service time* - the total elapsed execution time during realization of a response thread. This result represents the minimum achievable response time, given the service demands within the thread and the inherent service rates of the hardware devices used in fulfilling those demands. System overheads associated with intertask communication, communication protocol processing, transient error recovery, and other services are applied as appropriate in calculating thread service times.
- *Device utilization* - the total (offered) load placed on a hardware device. For each hardware device, this result is defined as the total of all service demands (per unit time) on the device divided by the device's service rate, and is expressed as a percentage. Values greater than 100% indicate an unstable situation in which a device has insufficient capacity to handle the load placed on it; in the long run, queues for such a device will grow without bound.
- *Thread availability* - the probability that a thread is completed successfully. Successful completion requires that operational hardware devices are available to accept all constituent service demands, and that no uncorrected transient errors or software failures occur during realization of the thread.

Additional performance and reliability results that are calculated, and which can be displayed at the user's option, include:

- *Function service time* - the total elapsed service time during completion of all service demands within a single primitive function in an SCF diagram.
- *Device availability* - the probability (at an arbitrary instant of time) that a particular device is in an operational state. When a device consists of a multi-unit subsystem, this availability result represents the probability that at least one of the units within the redundant configuration is available to accept arriving service demands.

Figure 4 shows an SCF diagram on which response thread service time, response thread availability, and function service time results are displayed. Figure 5 shows a hardware architecture diagram in which each device is annotated with device utilization and device availability results.

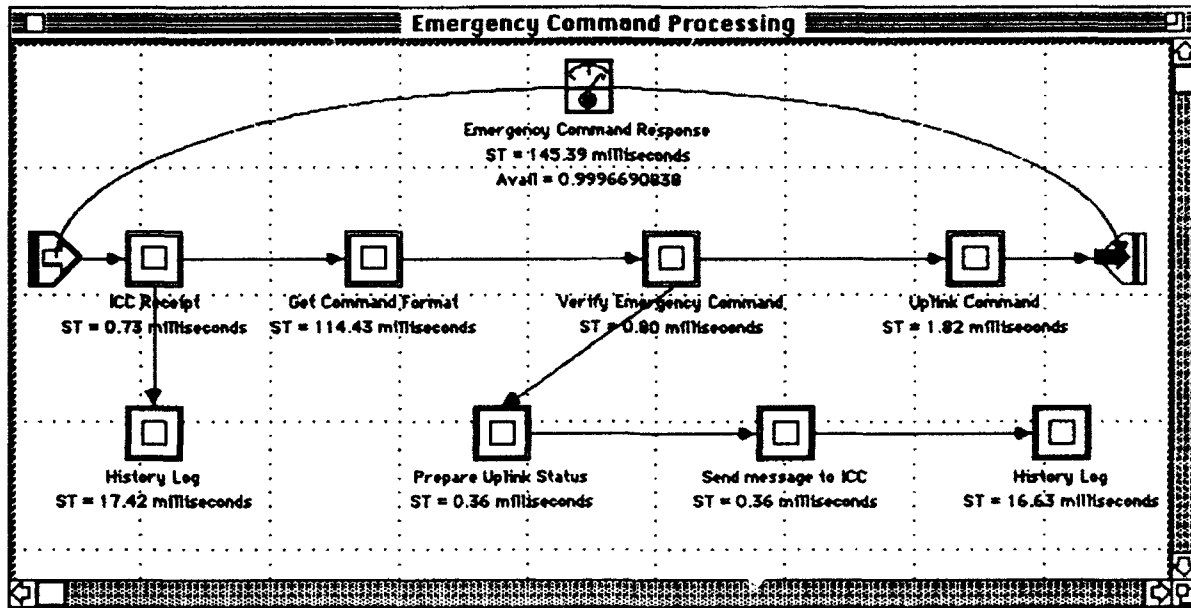


Figure 4: Thread Service Time and Availability Results

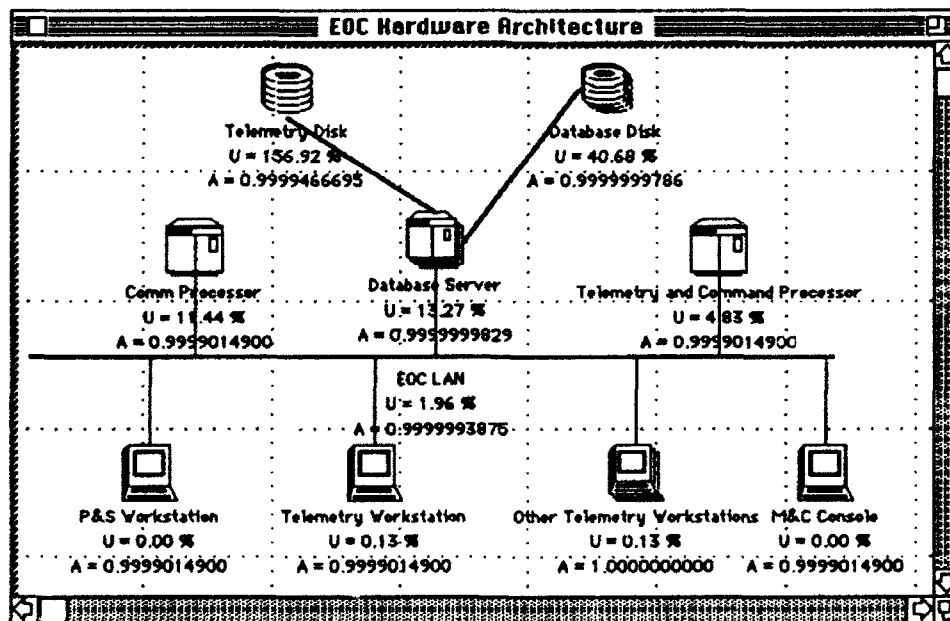


Figure 5: Device Utilization and Availability Results

Causally Factored Performance and Reliability Results

The automated reasoning component of START/ES requires not only top-level measures of performance and reliability (e.g., for assessing compliance with requirements), but also lower-level measures which indicate the contributions of individual design components and attributes to response thread delays and failure likelihoods. Such measures are needed during the problem identification reasoning process, which attempts to isolate the root causes of performance and reliability shortfalls. In order to obtain this information, *causally factored* performance and reliability data is collected during execution of the design evaluation algorithms.

Factored performance results. During performance computations, START/ES traces through control flow diagrams in order to accumulate the total service time within the scope of each response thread, and the total utilization of each hardware device. These accumulations are based on "atomic" service time events that are implied by individual service demands occurring within the control flows. For each atomic service time event encountered, a data record containing various numerical values, as well as qualitative information regarding the *context* within which the service time event occurred, is created and stored. When performance computations are complete, the set of all of these records serves as a data base of factored performance results, from which the contributions to thread service times and device utilizations associated with various design elements can be reconstructed during execution of the automated reasoning component.

Specific context elements attached to service time values include the dispatchable unit of software (task) within which the service time event occurred, and the type of application or system overhead operation being performed (e.g., instruction execution, intertask communication, intercomputer communication, remote data access, local data access). Due to the emphasis of START/ES on reasoning about performance/reliability tradeoffs, it is essential that processing delays directly related to the use of fault tolerance capabilities be identified; for example, the context element "transient error recovery" is attached to service times incurred during replicated operations associated with error detection and correction.

Figure 6 illustrates the performance data collection process. The information recorded for each service time event includes the service time value itself, the applicable flow arrival rate, the device on which the event occurred, and the attached context information. Note that the elements of context attached to individual service time values may be embedded within one another; for example a remote data access (a "GET" operation) which includes intercomputer communication (ICC) between two processors will have both the "GET" and "ICC" context elements attached.

The context-based recording scheme used for factored data collection provides a great deal of flexibility for generation of the basic facts upon which the expert system operates. New elements of context are easily accommodated, and since the context attached to each service time value is of arbitrary length, contextual information can be recorded to any level of depth.

Factored reliability results. Thread availability for a particular response thread is defined as the probability that all constituent service demands within the thread are successful, which in turn requires that all of the following are true: (1) all service demands are successfully accepted on an operational unit within a hardware subsystem, (2) no uncorrected transient errors occur during execution on any operational unit, and (3) no software failures occur during execution of any software module invoked by the thread.

During thread availability calculations, the following are recorded for each response thread: (1) the set of hardware subsystems required, (2) the probability of one or more transient errors occurring during execution on each subsystem unit actually used, and (3) the probability of one or more software failures occurring during execution of each software module invoked. This information, together with availability results calculated for each hardware subsystem, allow overall thread availability to be

calculated. At the same time, thread availability results are factored according to the relative contributions of the basic sources of functional failure, or "failure classes": *hardware unavailability* in hardware subsystems, *uncorrected transient errors* on active units, and *software failures* in software modules.

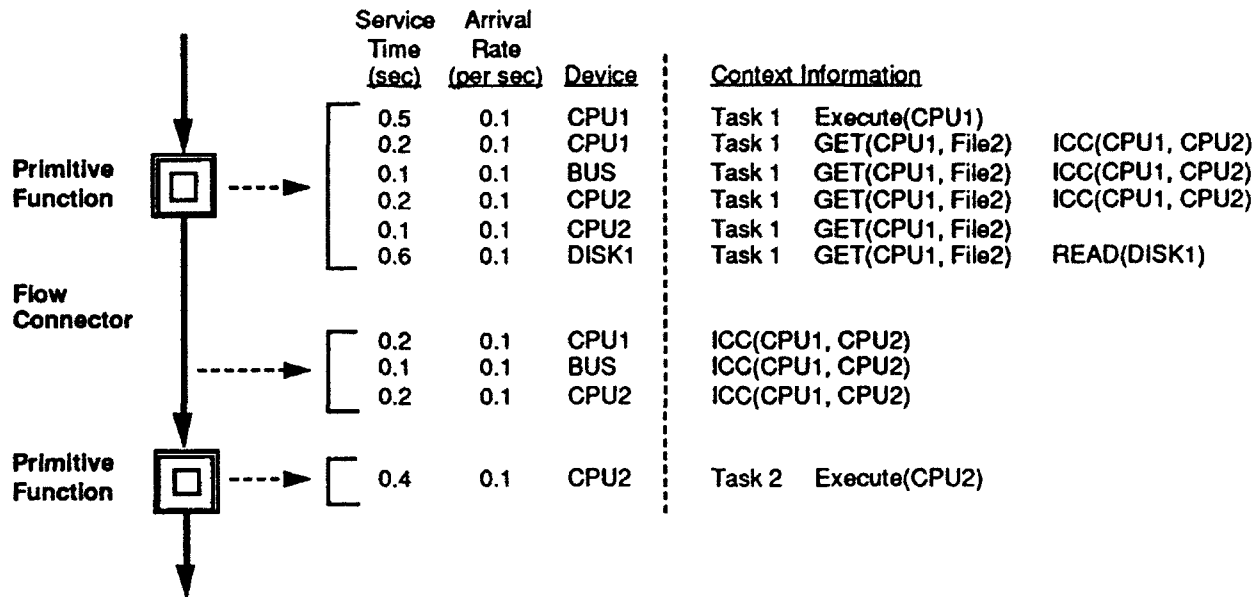


Figure 6: Context-Based Data Collection

The *unavailability contribution* of each failure class to the total unavailability of a given response thread is defined as the total amount of unavailability in the thread which could be reduced by decreasing occurrences of failures of that type, i.e. as the difference between the availability of the thread given that failures of that type do not occur, and the availability of the thread as calculated. These unavailability contributions are normalized so that the set of *contribution factors* assigned to the various failure classes sum to unity:

$$\text{Contribution factor (Class } i) = \frac{\text{Unavailability contribution (Class } i)}{\sum_j \text{Unavailability contribution (Class } j)}$$

Hardware unavailability results are further factored to quantify the contributions of three basic causes of hardware unavailability. *Maintenance downtime* is unavailability due to all units in a device subsystem being in repair. *Reconfiguration downtime* is unavailability due to switchover of backup units into active service. *Detection latency* is unavailability caused by attempts to assign service demands to a unit which has failed, but whose failure has not yet been detected by the system. The contribution factors assigned to each of these failure subclasses, as a proportion of the contribution factor assigned to hardware unavailability for a particular device subsystem as a whole, are determined from the steady-state solution to the mathematical availability model upon which subsystem availability results are based:

$$\text{Contribution factor (Subclass } k) = \frac{\sum_{j \in S_k} P_j}{(1 - A_S)} \times \text{Contribution factor (Hardware unavailability)}$$

where P_j = steady-state probability associated with subsystem state j

S_k = set of subsystem states corresponding to type k unavailability

and A_S = subsystem availability

5. Expert System

The START/ES automated reasoning component is implemented using the CLIPS expert system shell. System description and design evaluation results, represented as CLIPS facts, are queried during execution of the rule base, which is also encoded in CLIPS.

The rule base is partitioned into three rule sets. These rule sets are designed to be executed sequentially, so that the information generated during each reasoning stage can be accumulated and used during subsequent stages. The results of each stage are also displayed to the user.

Reasoning Stage 1: Compliance Assessment

Compliance assessment rules compare top-level performance and reliability results for each response thread to the corresponding user-specified requirement values. A system is asserted to be non-compliant if at least one response thread in the system is non-compliant. After all defined response threads have been checked for compliance -- which in a system containing many critical functions may involve a large number of comparisons -- a list of non-compliant threads is displayed to the user, as illustrated in Figure 7.

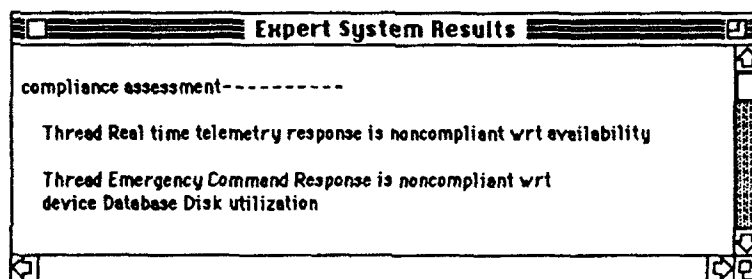


Figure 7: Compliance Assessment Results

A response thread is considered to be non-compliant with its performance requirements if either (1) the computed thread service time exceeds the thread response time requirement, or (2) at least one device or task used by the thread exceeds its resource utilization limit. In the first case, the thread is asserted to be *non-compliant with respect to service time*, and in the second case, the thread is asserted to be *non-compliant with respect to utilization* of the device or task. The reasoning behind this approach is that if thread service time exceeds the required response time, then the thread cannot possibly meet its response time requirement. However, if the service time is less than the response time requirement, the thread may still fail to meet its requirement -- due to contention effects -- if the utilization of one or more resources used during execution is high. This approach allows compliance with performance

requirements to be assessed using an analytic modeling approach which avoids explicit analysis of resource contention effects.

A response thread is considered to be *non-compliant with respect to availability* if the calculated availability of the thread -- indicating the probability that all service demands within the thread are completed successfully -- is less than the thread availability requirement.

Reasoning Stage 2: Problem Identification

For each non-compliant response thread identified during the compliance assessment reasoning stage, problem identification rules examine causally factored design evaluation results to determine the most significant contributors to performance and reliability shortfalls. Because the context information attached to causally factored results is of arbitrary depth and complexity, the number of different design elements which must be considered as potential contributors may be quite large. The ability to isolate system components and design features that are most critical in terms of meeting requirements -- from among such a large set of potential elements -- is perhaps the most useful aspect of the problem identification results provided by the expert system to the user (Figure 8).

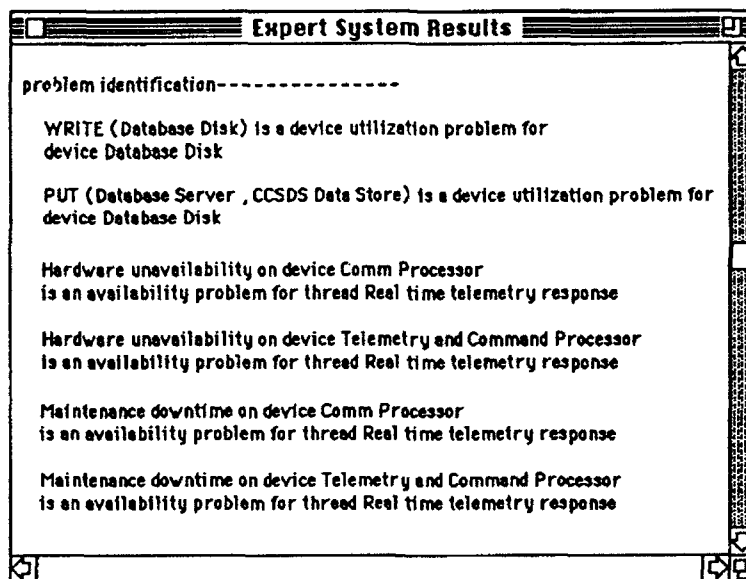


Figure 8: Problem Identification Results

Given that a particular response thread is non-compliant with respect to either service time or utilization of a resource, design elements associated with a large percentage of the amount of non-compliance are identified as *service time problems* or as *utilization problems*. Design elements which may be identified include individual hardware devices, fault tolerance overheads, and system service overheads such as intertask communication on a processor, intercomputer communication between two processors, remote data access between a processor and a data file, and local data access on a storage device.

Given that a particular response thread is non-compliant with respect to availability, failure classes which contribute most significantly to thread unavailability are identified as *availability problems*. The significance of failure class contributions is assessed based on the unavailability contribution factors computed for the thread. Failure classes which may be identified include hardware unavailability,

maintenance downtime, reconfiguration downtime, detection latency, or uncorrected transient errors on a hardware device, and software failures in a software module.

Reasoning Stage 3: Design Change Recommendation

Design change recommendation rules employ expert judgements in attempting to recommend design changes which will reduce the magnitude of one or more identified problems, and thus move the system in the direction of compliance with its requirements. A primary goal of the recommendations produced is to assist the designer in understanding the sensitive tradeoffs that attend various design choices, including the subtle interactions between performance and reliability behavior. Figure 9 illustrates a set of recommendations produced by the expert system; these recommendations include design changes in both the performance and reliability areas, and address inherent device characteristics as well as system-level architectural issues.

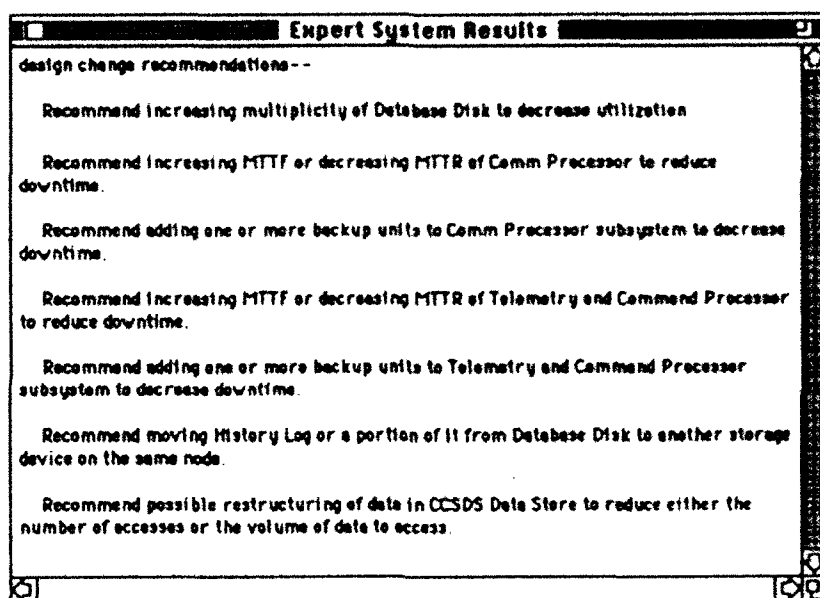


Figure 9: Design Change Recommendations

The design change recommendation rule base can be divided conceptually into six areas, each of which pertains to a particular set of system design issues:

- The *hardware characteristics* design area is concerned with inherent hardware device performance and reliability attributes. When performance problems have been traced to a particular device, rules in this area suggest improvements to such characteristics as device processing speed and multiplicity. Similarly, when reliability problems have been traced to a particular device, improvements to the basic device attributes Mean Time to Failure (MTTF) or Mean Time to Restore (MTTR) are recommended.
- The *task structure* design area is concerned with packaging of software into tasks, which are the lowest level of concurrency represented in START/ES. Rules in this area focus on such performance issues as the degree of parallelism that can be achieved through various partitioning schemes, versus the amount of intertask communication overhead incurred in each case.

- The *functional and data allocation* design area is concerned with the assignment of tasks and data files to hardware devices. Alternative assignment schemes may affect many different performance indicators, including thread service times, device utilizations, and intercomputer and intertask communication overheads. Task and file allocations may also affect reliability, in that alternative assignments may introduce different sources of unreliability into response threads which require the use of the devices to which tasks and files are assigned.
- The *hardware subsystem structure* design area is concerned with the performance and reliability implications of alternative hardware redundancy structures and redundancy management capabilities. Rules in this design area address the selection of subsystem design parameters -- including the levels of active and standby redundancy employed -- which will achieve the best overall balance between subsystem performance and reliability effects.
- The *transient error recovery* design area is concerned with the performance and reliability implications of alternative transient error detection and correction schemes. Rules in this area attempt to balance the amount of coverage provided for transient errors with the amount of additional processing time and resource utilization caused by replicated operations.
- The *software reliability* design area is concerned with the effects of inherent software module reliability on response thread unavailability.

As an example of design change recommendation reasoning, consider a hardware subsystem which has been identified as a utilization problem. If this subsystem contains at least one backup unit, then the effective load on the subsystem can be reduced by using one or more of these backups in active mode. However, doing so may increase the overall failure rate of the subsystem, and thus increase overall downtime. Thus, provided that maintenance downtime on the subsystem is not already an availability problem, it is recommended in this case that a backup unit be used for active processing:

IF	Device D is a utilization problem; and Maintenance downtime on Device D is not an availability problem; and Number of backup units in Device D subsystem ≥ 1 ;
THEN	Recommend changing one or more Device D subsystem backup units to active units to increase service capacity.

As another example of a design change recommendation rule, consider a device on which uncorrected transient errors are an availability problem. If there is only one active unit in the device subsystem (thus precluding the use of N-modular redundancy with the existing hardware), then it is recommended that a temporal repetition scheme be used to detect and correct errors. However, since using that scheme will significantly increase service times on the device, this recommendation is only made if the device is not already a service time problem:

IF	Uncorrected transient errors on Device D is an availability problem; and No error recovery scheme is selected on Device D; and Number of active units in Device D subsystem = 1; and Device D is not a service time problem;
THEN	Recommend using temporal repetition on Device D to reduce failures caused by uncorrected transient errors.

6. Rule Builder Interface

Recent experience with a rule-based expert system tool for performance analysis [6, 7] indicated that the utility of such a tool is greatly enhanced if the capability to specialize, extend, or otherwise modify the rule base is provided for the expert user. For START/ES, this capability is provided for the design change recommendation portion of the rule base. Access to these rules is provided by the rule builder interface.

The rule builder interface is based on an "English-like" *rule language* that allows rules to be expressed in a manner more natural to the expert than the pattern matching syntax used in CLIPS. The translation from rule language forms to internal CLIPS rule representations is handled automatically. Figure 10 illustrates the dialog used to edit rule clauses, in which "pop-up" menus are used to provide access to alternative selections for various rule components.

IF hardware unavailability on hardware device to be called
p is an availability problem for
thread to be called t

Done

Figure 10: Rule Editing Dialog

The rule builder interface is supported by various utilities designed to assist the expert user. These include the ability to search the rule base for specified character strings, the ability to create, load, and save alternative rule bases, and the ability to stop and resume expert system execution at the point at which a specified rule fires.

As an example of the way in which the expert reasoning capability may be extended, consider a situation in which several response threads do not meet their availability requirements, and in which several different devices have been identified as availability problems. Currently, the design change recommendation rule base will recommend that the inherent reliability of all of these devices be increased. A natural extension would be to identify devices whose unreliability is particularly problematic -- and which are therefore prime candidates for improvement. For example, a rule could be formulated which would find devices that are availability problems for *more than one response thread*, and recommend improving the reliability of these devices first, before considering those which affect only a single thread.

Conclusions

START/ES provides automated interpretation of performance and reliability model results. Using expert system technology, it identifies problems and recommends design changes. Initial evaluation with small scale system designs has shown the potential value of the expert system based approach. START/ES has been able to process causally factored model results and isolate the causes of non-compliance with performance and reliability requirements. Its recommendations, when interpreted by the user in the context of the current system design, provide valid guidance in converging on a fully compliant design.

At the same time, early use has revealed the need for further research and enhancements. In the area of system description, an ability to define specialized system design elements and their performance and reliability behaviors would be a valuable addition. In the area of performance models, a simulation capability is needed to evaluate contention effects more robustly than the current analytic model. In the area of the expert system, the rule builder needs to be extended to access all system description elements and model results. The rule language needs to be more comprehensive in terms of its logical expressiveness. Finally, START/ES needs to be evaluated on real-world problems of scale. These issues represent future work related to START/ES and, more generally, expert system applications to performance and reliability modeling.

Acknowledgements

The development of START/ES was sponsored in part by NASA - Goddard Space Flight Center, Greenbelt, Maryland. The development of the START tool upon which START/ES is based was sponsored in part by the Naval Surface Warfare Center - White Oak Laboratory, Silver Spring, Maryland.

References

1. E. Brehm and R. T. Goettge, "START - A Tool for Integrated Performance and Reliability Evaluation," *Proceedings of the 1991 System Design Synthesis Technology Workshop*, Naval Surface Warfare Center, September 1991.
2. CLIPS Reference Manual, CLIPS Version 5.0, Software Technology Branch, Johnson Space Center, National Aeronautics and Space Administration, January 11, 1991.
3. J. B. Dugan, "On Measurement and Modeling of Computer Systems Dependability: A Dialog Among Experts," *IEEE Transactions on Reliability*, Vol. 39, No. 4, October 1990, pp. 506-510.
4. H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach," *IEEE Micro*, February 1989, pp. 25-40.
5. J. Arlat, K. Kanoun, and J. Laprie, "Dependability Modeling and Evaluation of Software Fault-Tolerant Systems," *IEEE Transactions on Computers*, Vol. 39, No. 4, April 1990, pp. 504-513.
6. R. Goettge, "PEDAS: An Expert System for Performance Engineering of Time-Critical Software," *Proceedings of the Computer Measurement Group*, December 1990.
7. R. Goettge, E. Brehm, and W. McCoy, "A Knowledge-Based Approach to Automated Interpretation of Performance Model Results," *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, Vol. I, January 1992, pp. 225-234.

Formalizing the Transition from Specification to Design for Real-Time Systems*

Armen Gabrielian
UniView Systems
1192 Elena Privada, Mountain View, CA 94040
armen@well.sf.ca.us

Abstract

An overview of a formal approach is presented for mapping a specification of a real-time system onto a design space of abstract "locations," consisting of hardware, software, communication components and human interfaces. The specification is assumed to be given in terms of a "hierarchical multi-state (HMS) machine," which is obtained by integrating an advanced type of state model with an interval-based temporal logic. Formal verification techniques and correctness-preserving or partially correctness-preserving transformation methods are two of the promising approaches for maintaining properties of a specification in the transition to design. The location concept also provides the means for deriving data exchanges and performance requirements on the design components of a system as progress is achieved towards implementation.

Keywords — Real-time systems, specification, design structuring, allocation of requirements, system modeling.

1 Introduction

While numerous specification and design methods for real-time systems have been investigated in the recent past, little progress has been made in formalizing the transition from specification to design in a way that guarantees the preservation of temporal properties. Standard design methodologies for real-time systems (e.g., [HP87, WM86]) depend entirely on informal methods, usually based on finite-state machine extensions of data flow diagrams. Since finite-state machines easily become intractable for even relatively simple systems, such representations are usually confined to the definition of simple local control conditions, with

*This work was supported in part by the Office of Naval Research under contract N00014-92-C-0047.

no hope of offering even a simulation-based analysis of behavior. Other methods such as DCDS [Al85] have addressed the simulation aspects, while various *formal* methods for specification and verification of real-time systems have been proposed in the literature (see, e.g., [Ga91a] and accompanying articles). Executable formal specification methods usually also provide simulation capabilities. However, it is well-known that simulation is simply inadequate for guaranteeing correctness.

The key stages in the transition from specification to design are (1) the structuring or partitioning of the specification space, (2) the definition of a design space, (3) the definition of a mapping from the specification space to the design space, and (4) the formal derivation of design requirements from the specification requirements. Success in the development of a formal basis for these four stages depends on the ability to define precise mathematical formulations of the specification and design spaces and the mappings between them.

The purpose of this paper is to present some preliminary ideas on the use of the "hierarchical multi-state (HMS) machines" [GF88, GI90, GF91, GI90, Ga91a] in formalizing the transition from specification to design. HMS machines are obtained by integrating parallel and hierarchical automata with a temporal interval logic, called TIL, to provide a formal methodology for specifying the behavior and requirements of hard real-time systems. Several independent formal methods for verifying "safety properties" of HMS machine specifications have been developed so far. The correctness-preserving transformations of [FG89] and the model-based theorem proving of [GI91, Ga91b] provide refutation based verification capabilities that, in general, avoid the need for complete enumeration of behavior. The model checking of [GI92] and the interacting computation graphs of [GI91] offer manageable approaches to enumerative verification.

To deal with the transition to design, we propose the partitioning of the state space of an HMS machine representing the behavior of a system into a set of "locations." A location is an abstraction of a piece of hardware, a software program, a communication medium or a human interface. Since an implementation creates its own requirements beyond the system requirements, a process of refinement is usually necessary that expands the specification and creates an extension of the original HMS machine. In addition, two other elements of an HMS machine must be mapped into the space of locations: (1) "transitions" that define *what* changes in states can occur in an HMS machine, and (2) "controls" consisting of TIL predicates that define *constraints* on transitions. The derivation of data interchanges and temporal requirements on the design space is derived essentially automatically once the space of locations is defined. The key problem that remains is to verify that the refinement satisfies the requirements. This can be accomplished in several ways. One approach is to employ transformations such as those in [FG89, GI92] that preserve or partially preserve behavior. Another method is to verify formally that key safety properties are preserved in the refined machine using one of the available verification methods for HMS machines. A third approach not considered here, which is commonly used in the context of process algebra, is bisimulation, in which one attempts to prove that the two specifications have identical behaviors.

In Section 2 of this paper we present an overview the basic concepts of HMS machines

and in Section 3 we employ the specification of a simple railroad operation to give an outline of our approach for transitioning a specification to a design. In the process, we also present our visual notation for representing HMS machines. In Section 4 we offer a brief summary and the conclusions.

2 Background and Definitions

We begin by defining the non-hierarchical version of a specification formalism that integrates a parallel version of automata with an interval-based temporal logic. Formally, we define a discrete-time, boolean "multi-state (MS) machine" as a triple $H = (S, \Gamma_D, \Gamma_N)$, where

1. S is a set of "states," any number of which may be true or "marked" at a moment of time.
2. Γ_D and Γ_N are, respectively, the sets of "deterministic" and "nondeterministic" transition of H . Each deterministic or nondeterministic transition is of the form

$$(\text{PRIMARIES}) (\text{CONTROL}) \rightarrow (\text{CONSEQUENTS}),$$

where $\text{PRIMARIES} \subseteq S$, $\text{CONSEQUENTS} \subseteq S$ and CONTROL is a predicate on the history of the states, expressed in a temporal interval logic called TIL. For a transition u , each state in the associated PRIMARIES (CONSEQUENTS) set of state will be called a "primary" ("consequent") state of u . Also, the predicate CONTROL for u will be called the "control" or "control predicate" of u .

3. A transition is "enabled" if its primary states are all true and also its associated control predicate is true.
4. At each discrete moment of time all the enabled deterministic transitions and a subset of the nondeterministic transitions "fire," causing changes in the marking of the states.

The set of hierarchical MS machine or "HMS machines" is the superset of the set of MS machines when some of the states are replaced by HMS machines. The details will not be presented in this paper. For definitions of recursive hierarchies and the use of different granularities of time at different level of hierarchy, see [Ga91a, GI92]. Extensions involving non-boolean states that accommodate data flows can be found in [GI90].

The behavior of a real-time system can be specified in terms of an HMS machine by representing its attributes as hierarchical states, with the control predicates defining the logical and temporal constraints under which changes in the system occur. We note that in an HMS machine many states can be true at a moment of time. In general, this results in significant reduction in the number of states compared to traditional finite-state machines.

We now present a notation and formal definitions for the temporal interval logic TIL and the state updating rule for HMS machines.

Notation Given an HMS machine H with state set S , the “marking” of H at time t is a mapping $M_t : S \rightarrow \{F, T\}$ that defines the set of marked or true states of H .

Definition 1 Given a marking M_t of an HMS machine at time t and a formula ψ , we denote the satisfiability of ψ in M_t by $M_t \models \psi$. The temporal interval logic TIL is then obtained by extending propositional logic with the following four operators:

- $O(t')$ At relative time t'
 $M_t \models O(t')\psi \Leftrightarrow M_{t+t'} \models \psi$
- $[t_1, t_2]$ Always between t_1 and t_2
 $M_t \models [t_1, t_2]\psi \Leftrightarrow \forall t' \ t_1 \leq t' \leq t_2 \text{ implies } M_t \models O(t')\psi$
- $\langle t_1, t_2 \rangle$ Sometime between t_1 and t_2
 $M_t \models \langle t_1, t_2 \rangle \psi \Leftrightarrow \exists t' \text{ such that } t_1 \leq t' \leq t_2 \wedge M_t \models O(t')\psi$
- $\langle t_1, t_2 \rangle!$ Sometime-change between t_1 and t_2
 $M_t \models \langle t_1, t_2 \rangle! \psi \Leftrightarrow \exists t' \text{ such that}$
 $((t_1 - 1) \leq t' < t_2) \wedge (M_t \models O(t')\neg\psi) \wedge (M_t \models \langle t' + 1, t_2 \rangle \psi).$

Definition 2 For each state s in an HMS machine, let $\Gamma_{in}(s)(\Gamma_{out}(s))$ be the set of transitions into (out of) s . Then, the marking of s at the next moment ($t = 1$), given the marking at the current moment ($t = 0$), is defined as follows:

$$O(1)s \Leftrightarrow (s \wedge (\bigwedge_{u \in \Gamma_{out}(s)} O(1)\neg u)) \vee (\bigvee_{v \in \Gamma_{in}(s)} O(1)v),$$

where for a function ψ , $\bigwedge_{x \in X} \psi(x) = T$ if $X = \{ \}$ and $\bigvee_{x \in X} \psi(x) = F$ if $X = \{ \}$.

Intuitively, a state s is true at time $t = 1$ if and only if (1) s is true at time $t = 0$ and no transitions fire out of it at $t = 1$, and/or (2) some transition fires into s at time $t = 1$.

3 Transition of a Specification to Design

In this section, we present an example of an HMS machine specification of a simple railroad operation and we provide an outline of a partial transition of some of its components to a design. The process requires two steps. In the first step, two sections of this specific HMS machine are mapped to a location space consisting of two components: (1) a software process that monitors a clock and sends a green light signal for a new train to start on the track, and (2) a mechanical device that operates a gate mechanism. In the second step, the section of the HMS machine specification for each location is refined to reflect design considerations. We note that the two steps can be reversed, i.e., it is possible to perform the refinement first and then to define the mapping to the location space. In fact, more choices for design are possible in the latter case.

Figure 1 presents our graphic notation for an HMS machine representing the operation of a simple railroad. In our notation, boxes represent states, dark arrows denote transitions, with an asterisk indicating that the transition is nondeterministic, thin arrows represent

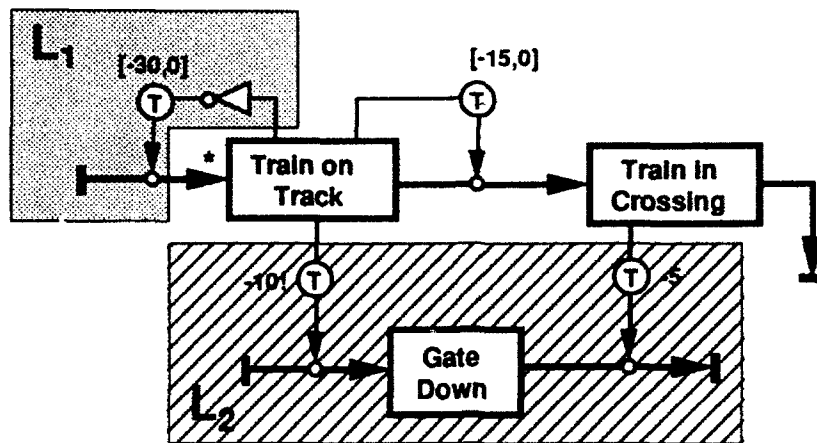


Figure 1: HMS Machine Specification of a Simple Railroad Operation

controls, and temporal operators appear next to the symbol \textcircled{T} . VLSI notation is used to form logical combinations of control predicates and a TIL predicate of the form $\langle t, t \rangle$ is abbreviated as t and a predicate of the form $\langle t, t \rangle!$ is abbreviated as $t!$. Also, a short thick line at the beginning (end) of a transition denotes the special state that is always true (false). Thus, the nondeterministic transition into the state "Train on Track" *may* fire if that state has been false continuously for the last 30 units of time. Also, the transition into the state "Gate Down" will fire if the state "Train on Track" was false and *became true* 10 units of time ago.

In Figure 1, a partial allocation of the states, transitions and controls of the HMS machine into the two locations L_1 and L_2 is made, as indicated by the shaded regions. In Figure 2, L_1 and L_2 are refined to reflect the transition of two parts of the specification to a simplified design. As mentioned earlier, we can assume, for example, that L_1 is to be implemented in terms of a software process that monitors a clocks and sends a signal to a green light to allow a new train on the track 30 time units after the previous train departs. As in Figure 1, the actual arrival is left nondeterministic as indicated by the asterisk next to the transition into the state "Start Train." The location L_2 may correspond to a mechanical device that starts a mechanism for lowering the gate 2 time units after the state "Train on Track" becomes true. The process of lowering the gate takes 7 time units. Five time units after the train passes the crossing, the gate is raised automatically so that it is no longer in the "Gate Down" position.

It should be noted that in Figures 1 and 2 we employ a discrete-time version of HMS machines, in which transitions fire at discrete integer-valued moments of time. This is consistent with the usual finite-state machine modeling approach to representing behavior. Under this assumption, it is easy to prove that time delays in location L_1 and L_2 of Figure 1 are maintained accurately in the respective locations in Figure 2. Thus, the time delay *budget* of 10 time units for the gate to be down in Figure 1 is allocated to two separate delays plus an extra transition in Figure 2. With the use of continuous-time HMS machines

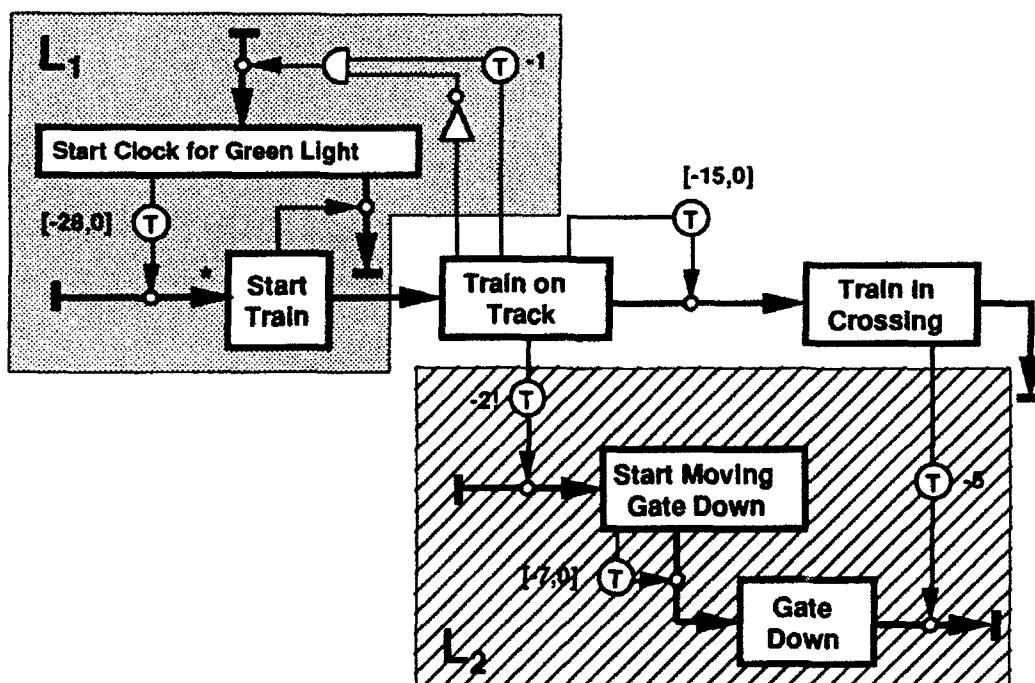


Figure 2: Partial Design of Railroad Operation

[Ga91b], the extra time for the second transition would not be required, resulting in a simpler demonstration of faithfulness of the refinement with the original specification. In both the discrete and continuous cases, the TIL language can be used to formally state the firing conditions for transitions in a specification and its refinement. Safety properties can also be independently verified for a design that is derived from a specification.

Transformations on specifications that maintain temporal properties, such as those in [FG89, GI92], offer another promising approach to transitionising a specification to design in a way that guarantees to maintain requirements satisfied by the specification. However, as noted by a number of writers, this is often not necessary. For example, one of the transformations in [FG89] only partially preserves behavior. In a possible application of such a transformation to our railroad example, one could investigate the design of the software under the assumption that the train starts immediately after the green light is turned on. Understanding of the environment is often necessary to determine the usability of transformations that do not strictly maintain behavior.

We have considered in this example a single-step refinement of a specification that maintains logical and temporal properties. The allocation of the refined specification to a set of locations can be considered as one step in the evolutionary process of design. Normally, a number of repeated refinement are necessary to reach a final design. To consider complex data flows, an extended version of HMS machines [GI90] can be employed that uses non-boolean states and replaces TIL with its first-order counterpart. For many commonly-occurring systems, however, the boolean version presented in this paper is quite adequate.

For example, in the present example, the boolean case is sufficient in identifying information flow into a location as the set of control arrows that enter its boundaries.

4 Summary and Conclusions

In this paper, we presented a brief overview of the hierarchical multi-state (HMS) machine specification methodology and demonstrated through an example the process of refining a specification to a design. The important advantages of our approach are: (1) hardware, software, communication elements and human interactions can be treated in a uniform manner, (2) formal verification can be used to assure the correctness of the original specification, (3) preservation of logical and temporal properties during the transition to design can be demonstrated by either a formal verification process or by limiting refinements to transformation that preserve or partially preserve behavioral properties, and (4) executability of the visually-expressed HMS machine formalism provides further analysis capabilities in either simulating behavior in the forward direction or detecting causes of errors by simulation in the *backward* direction. A branching backward simulation is, in fact, the basis of the model-based theorem proving of [Ga91b, GI91].

References

- [Al85] Alford, M., "SREM at the age of eight; the Distributed Computing Design System," *Computer*, April 1985, pp. 36-46.
- [FG89] Franklin, M.K., and A. Gabrielian, "A transformational method for verifying safety properties in real-time systems," *Proc. 10th Real-Time Systems Symposium*, Santa Monica, CA, December 5-7, 1989, pp. 112-123.
- [Ga91a] Gabrielian, A., "HMS machines: a unified framework for specification, verification and reasoning for real-time systems," *Foundations of Real-Time Computing: Formal Specifications and Methods*, A. van Tilborg and G. Koob (Eds.), Kluwer Academic Publishers, Norwell, Mass., 1991, pp. 139-166.
- [Ga91b] Gabrielian, A., "Verifying safety properties of HMS machine specifications by (inductive) theorem proving (extended abstract)," *Proc. Fourth Annual Review and Workshop, Foundations of Real-Time Computing Research Initiative*, Washington, DC, Oct. 31 - Nov. 1, 1991, pp. 111-116.
- [GF88] Gabrielian, A., and M. K. Franklin, "State-based specification of complex real-time systems," *Proc. IEEE Real-Time Systems Symposium*, Huntsville, AL, 1988, pp. 2-11.
- [GF91] Gabrielian, A., and M.K. Franklin, "Multi-level specification real-time systems," *Communications of the ACM*, Vol. 34, No. 5, May 1991, pp. 50-60. Earlier version

in *Proc. 12th International Conference on Software Engineering*, March 26-30, 1990, Nice, France, pp. 52-62.

- [GI90] Gabrielian, A., and R. Iyer, "Specifying real-time systems with *extended* hierarchical multi-state (HMS) machines," Technical Report No. 90-21, Thomson-CSF, Inc., Palo Alto Reserch Operations, January 1990.
- [GI91] Gabrielian, A., and R. Iyer, "Verifying properties of HMS machine specification of real-time sytems," *Proc. Third Workshop on Computer-Aided Verification*, Aalborg, Denmark, July 1-4, 1991, pp. 489-500. Revised version to appear in Springer-Verlag LNCS series.
- [GI92] Gabrielian, A., and R. Iyer, "Integrating automata and temporal logic: a framework for specification of real-time systems and software," in *The Unified Computation Laboratory*, C.M.I. Rattray and R.G. Clark (eds), Institute of Mathematics and Its Applications, Oxford University Press, 1992, pp. 261-273 (to appear).
- [HP87] Hatley, D.J., and I.A. Pirbahi, *Strategies for Real-Time Systems Specification*, Dorset House, New York, 1987.
- [WM86] Ward, P.T., and S.J. Mellor, *Structured Development for Real-Time Systems*, Yourdon Press, Vol. 1 (1985) and Vol. 2 (1986).

Cooperative resource management in R-Shell

Wei Zhao

Swaminathan Natarajan

Department of Computer Science, Texas A&M University
College Station, TX 77843-3112

Abstract

In most real-time systems, the OS and the application share the responsibility for resource management, with each having its own well-defined role in the resource management process. They act as separate units, rather than co-operating to exploit the knowledge of each or jointly implementing the desired functionality. We propose an approach based on the concept of scheduling agents, which reside in the application run-time environment, and are customized to provide just those resource management functions which are needed by the specific application. The scheduling agent implementation is customized to the particular OS and system configuration, thus exploiting OS level knowledge. With this approach, we avoid the need for a sophisticated OS which provides a variety of generalized functionality, while still not burdening application programmers with heavy responsibility for resource management. We use scheduling agents as the basis of a flexible, fully distributed, object-oriented framework called R-Shell for building real-time applications which can adapt to dynamic variations in resource requirements and resource availability. In addition to scheduling agents, R-Shell includes an object-oriented OS, and the concept of an object-oriented resource hierarchy to describe resource requirements and resource characteristics, and supports a new technique called resource substitution.

1 Introduction

A hard real-time system has a dual responsibility of not only producing correct results, but also meeting application deadlines while producing these results [15, 3, 11]. Thus, there are two correctness properties which must be satisfied: *functional correctness* and *resource correctness*. Further, real-time and fault-tolerant systems have the unique characteristic that the correct (timing) behavior of the system is affected by the availability of all the computational resources which are needed. If any resource, such as the processor, memory, disk drives, files, network etc. are even temporarily unavailable, then the program may not complete before the deadline, or may not produce correct results. Thus, *timing correctness* can be generalized to the problem of *resource correctness*. The need to attain timing and resource correctness while maintaining functional correctness makes the design of real-time fault-tolerant systems a particularly difficult problem.

The keys to achieving timing and resource correctness are the concepts of *predictability* and *guarantees*. In order to ensure that enough time and resources will be available to applications, we must be able to determine their resource requirements in advance, and then set up resource allocation strategies so that all of the requirements are met. To obtain predictability, we must use design techniques which make it easier to determine resource requirements either analytically or empirically, at compile-time. Once

the resource requirements are known, the operating system must be designed so that it can provide guarantees to applications that the needed resources will indeed be available.

Much of the current research in real-time systems focuses on these two problems, with language and applications designers working on building systems with fixed resource requirements, and operating systems designers working on developing scheduling algorithms for allocating resources so that tasks with known resource requirements can meet deadlines. With this approach, the burden of ensuring timing and resource correctness at run-time is placed entirely on the OS. This is a reasonable approach if the application has a fixed set of functions to perform, operates in a stable environment, and is primarily repetitive in nature, such as signal processing applications. However, it does not work so well if the resource requirements of tasks may vary at run-time, or if resource availability can change during execution due to the occurrence of faults, or due to preemption of resources by higher priority tasks.

On the other hand, some systems use models where the responsibility for meeting deadlines is placed on the application, with the OS providing some support capabilities which the application can utilize. This approach can deal with resource scarcity situations by providing appropriate exception handlers in the application. The difficulty is that this burdens the application programmer heavily. Moreover, this approach is feasible only if there are a relatively small number of situations which the application must deal with, otherwise providing handlers for all combinations of possible situations becomes intractable.

However, a large, complex, and highly dynamic application such as an aircraft control system may have a variety of timing and resource requirements to meet, and these may change constantly. The resources available to the OS may also change as the operating environment changes. To meet this challenge of handling a wide variety of complex dynamic situations involving resources, it is necessary that the OS and the application should work together and ensure that the application resource requirements match the actual current availability of computational resources. Our design of the R-Shell run-time support system is aimed at integrating the design of applications with OS capabilities, and at facilitating run-time co-operation between application and OS.

The R-Shell system is based on the concept of *scheduling agents*, which interface between the application and the OS, and perform resource management functions. Scheduling agents are part of the run-time environment for a particular application, and are generated at compile-time to provide exactly that functionality which is needed by the application. They reside on top of an object-oriented OS. They can be designed to provide any resource management functionality appropriate to the application, such as obtaining guarantees from the OS that a method invocation in an object-oriented real-time application will have enough resources to meet its deadline.

In this document, we describe the motivation for the R-Shell approach, and our design of the R-Shell system. The rest of this section describes the complexity of the problem of building distributed fault-tolerant real-time systems. Section 2 presents a characterization of alternative approaches which may be employed to address this problem, and where several of the existing systems fit into this characterization. In section 3, we present our design of the R-Shell system. Section 4 summarizes the discussion.

Problem description: The complexity of large real-time systems

A large real-time system typically consists of many tasks, each with their own individual requirements. Moreover, in applications such as aircraft control, these tasks may have substantial inherent unpredictability. There may be several application characteristics which complicate the process of resource allocation in the system:

- **Deadlines:** The resource allocation process must ensure that deadlines of critical operations are met. Failure to meet these deadlines may have catastrophic consequences for the system,
- **Periodic tasks:** The application may include some functions such as data acquisition and monitoring which must be performed repeatedly.
- **Aperiodic tasks:** User queries as well as unexpected external events can both give rise to non-periodic requests. The OS must service these requests without jeopardizing the deadlines of critical tasks.
- **Precedence and grouping constraints:** There may be relationships among different tasks constituting an application: they may share data, or one may process data produced by another. In addition to these synchronization and precedence requirements, there may be grouping constraints among several tasks which cooperate to perform a function, so that their result is useful only if all of them complete.
- **Resource usage:** Some tasks may require resources with specific characteristics, or may need varying amounts of resources in different executions. For example, a particular task may need a 32-bit processor with a floating point accelerator in order to produce precise and timely results. The processing and network bandwidth of a radar system may depend on the number of targets being tracked currently.
- **Support for application-specific techniques:** The application may use particular techniques such as recovery blocks [9], and imprecise computation [5] to handle specific situations involving scarcity of resources. The system designers need to ensure that the OS resource management policies are compatible with the various application-specific techniques.

In addition to this complex application model, the computational environment may itself have several characteristics which make resource management still more difficult:

- **Faults:** The requirements for fault-tolerance and degraded modes of operation imply that resource availability in the system may vary in different situations, and applications should be able to adapt their behavior to match the changing resource availability.
- **Resource limitations:** Even under normal operation, the limited availability of resources may create a problem if application resource needs vary. For example, the network may become a bottleneck if the radar system needs to track many different targets.
- **Dynamics:** The resource characteristics may vary as application characteristics vary. For example, network behavior changes under different levels and different types of network load, making it difficult to achieve predictability.
- **Emergencies:** When emergency situations occur, resource allocation must be modified to devote maximum resources to emergency handling. This requires that the scheduling of all resources, including networks, should be based on preemptive priority-driven schemes.

We can divide this bewildering variety of requirements into four categories of problems which the resource management strategy must address:

1. Scheduling and resource allocation to meet deadlines, and to handle precedence and grouping constraints for periodic and aperiodic tasks.

2. Obtaining and using semantic information about applications, including variations in resource requirements, and needs for specific resources.
3. Handling of faults and emergencies.
4. Providing support for application-specific techniques.

OS design for fault-tolerant, distributed, real-time systems is an extremely complicated problem, since it must cater to a variety of application needs in a highly dynamic computational environment. It is our belief that it is difficult to handle this complexity either purely in the OS or purely in the application that motivates our approach of cooperation between application and OS in addressing these issues.

2 Classification of current approaches

Conventionally, issues relating to resource management are handled entirely in the OS. It is considered desirable to free the application from the burden of worrying about computational resources. However, this is typically not possible in a dynamic real-time system. Only the application may have knowledge of variations in resource requirements. The handling of faults and overload situations may be application-specific. Application semantic information may be needed for the scheduler to ensure that deadlines will be met. For all these reasons, the responsibility for resource management, and by extension the handling of dynamic behavior, is usually shared between the application and the OS. We characterize the approaches to addressing the problem based on the degree to which each of them share this responsibility.

There is a spectrum of possible approaches, ranging from handling the problem entirely in the OS, to handling it entirely at the application level. The variation in the division of responsibility is really a continuous one, so that we cannot provide a strict enumeration of the different possibilities. Nevertheless, we arrive at a broad classification of the solutions into four approaches, which correspond to basic differences in design philosophy. Figure 1 illustrates this classification pictorially.

The following is our characterization of the solution approaches:

- **Application-controlled:** At one extreme is the situation where the OS does not provide any special support whatsoever. The application must incorporate all the techniques needed to handle dynamic situations. This is the default in current practice.
- **OS-controlled:** The other extreme is the situation where the OS takes all the responsibility for performing scheduling to ensure that deadlines are met, without any input from the application. While this is convenient from the viewpoint of the application designer, it is inherently limited in terms of the issues which it can address. In particular, it cannot deal well with unpredictable variations in application resource needs, and does not support application-specific techniques (which might produce better results).
- **OS-controlled using application semantics:** This approach puts the bulk of the responsibility on the OS, with the application supplying it with semantic information, such as dynamic deadlines, resource requirements, criticality and value function information. This approach is more powerful than the previous one, and still does not place a heavy burden on the application. Like the previous approach, it requires a sophisticated OS design.

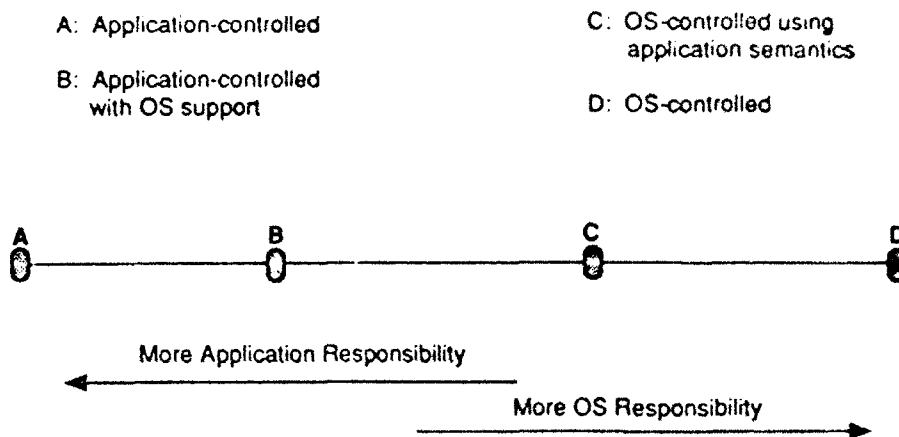


Figure 1. Spectrum Of Approaches To Handling Dynamic Behavior

- **Application-controlled with OS support:** This approach lets the application do the work of dealing with dynamic behavior, but the OS contains features and mechanisms with which the application can obtain status information and modify system behavior. For example, the OS may provide information about resource availability and current resource usage by other applications, and allow applications to customize the scheduling policies and modify resource allocations. With this approach, applications have considerable flexibility in terms of implementing different techniques to deal with situations.

In [8], we discuss several current systems, including ARTS [13], CHAOS [2], Concord [5, 7], GARTEN [10, 6], MARUTI [4] and Spring [12, 14], and where they fit into this characterization.

3 The R-Shell system

We propose a system called **R-Shell** that provides run-time support for large, complex, fault-tolerant distributed real-time applications. R-Shell consists of an object-oriented OS, object-oriented applications, and scheduling agents which interface between the applications and the OS. The design of R-Shell is based on the approach of co-operative resource management. In terms of the classification presented in Figure 1, co-operative resource management does not correspond to a particular point on the spectrum. Instead, it spans the entire spectrum, and individual applications designers can define the respective roles of OS and applications based on the needs and characteristics of the application.

In this approach, the OS and the application share the responsibility for resource management, based on some systematic design methodology. It must be emphasized that this approach is not the same as an equal division of responsibility; there is an associated design methodology using which application designers set out the roles of the application and the OS, and decide the modes in which they interact

and co-operate to achieve the overall goal. The power, flexibility and utility of this approach depend on the methodology itself. A good methodology can lead to a well-integrated design where the application and the OS dovetail perfectly, taking responsibility for just those aspects of the overall behavior and functionality which they are best-equipped to handle.

In R-Shell, the *scheduling agents* which interface between the application and OS are constructed to fit the needs of particular applications. The OS capabilities they utilize and the functionality which they provide to the application can both be determined by applications designers based on the implementation platform and application requirements. However, the scheduling agents are not a part of the application. They are part of the run-time support system provided by the software development environment.

3.1 The R-Shell framework

The R-Shell system is based on an open system concept, that each application should be able to obtain, and pay the performance penalty for, exactly those elements of functionality which it requires. The problem with incorporating many features in the OS is that the OS becomes very complex, and also quite slow, which is unacceptable in some real-time applications. On the other hand, if the application must incorporate many different resource management techniques, there is a heavy burden on the applications programmer, and also there may be a performance loss since resource management is not tailored to the particular system configuration. Run-time environments (RTEs), on the other hand, are developed for particular implementation platforms, and moreover, compilers ensure that RTEs contain just those elements of functionality which a particular application needs. Therefore, we believe that RTEs are the ideal components for incorporating sophisticated resource management features which may be needed only by some applications. Moreover, since RTEs are generated in the context of a particular application, they can utilize application semantics to implement application-specific features. Hence the key feature of R-Shell is the use of *scheduling agents* which are part of the run-time support environment, and interface between the application and the OS.

It should be noted that including resource management functions in the RTE is not in itself a novel idea. Conventional RTEs do manage memory with heaps and stacks. The Ada RTE does perform scheduling functions for tasks within an Ada application. The novelty of our approach is in the use of scheduling agents in the RTE to take advantage of both semantic and configuration information, in giving application programmers control over the functionality provided by the agents, and in the extensive capabilities and responsibility which we propose for scheduling agents.

Figure 2 shows the structure of the R-Shell system. The RTE for each application includes a scheduling agent which performs resource management functions for that application. Each type of physical resource has a resource manager which schedules use of resources of that type. The scheduling agent interacts with the resource managers for each individual resource to obtain all the resources needed by an application. Resource managers interact with each other to coordinate the allocation of resources to different applications.

Scheduling agents can perform a variety of different resource management functions, depending on the needs of the application, and the facilities provided by the underlying OS. Throughout the rest of this document, we illustrate the use of scheduling agents by describing some typical functions needed in a real-time system. In particular, real-time systems need to *express resource needs*, *obtain guarantees about resource allocation*, and *handle exception situations* where resources suddenly become unavailable due to faults or preemptions. We will discuss the concepts of R-Shell by describing an OS which provides guarantee and exception notification features, and an application which needs to express its

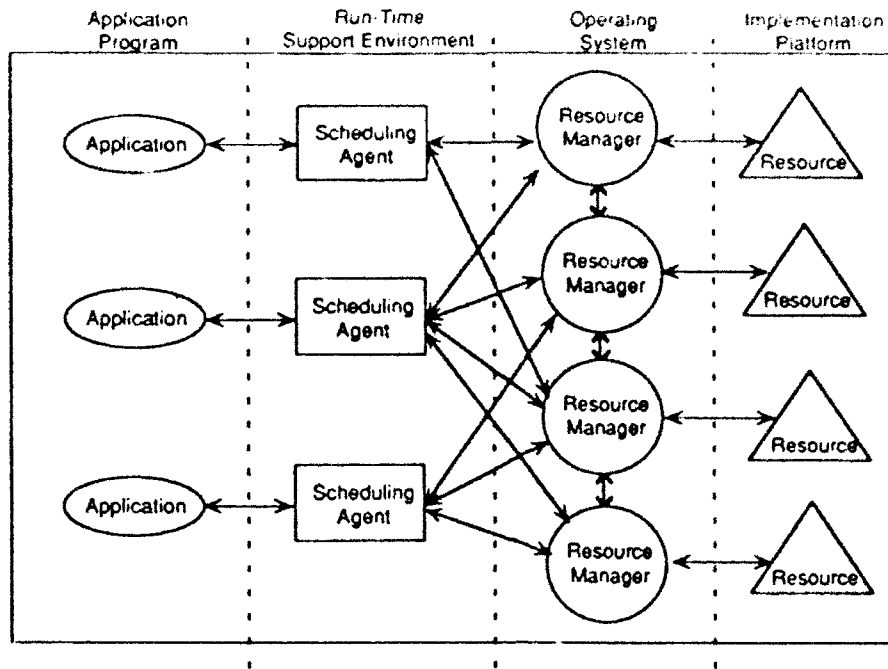


Figure 2: The R-Shell System

resource requirements, to obtain guaranteed resource allocation, and to handle resource exception situations. However, it should be noted that scheduling agents can be designed to implement any resource management functionality which is appropriate to the needs of a particular application.

The R-Shell approach represents an integration of the functionality of real-time applications and of the OS, with respect to resource management. This integration is accomplished by the use of scheduling agents. Conventionally, the OS is a configuration-dependent entity, which coordinates the allocation of resources, and handles situations such as faults using general techniques which are independent of application semantics but may be dependent on resource characteristics, such as process migration, message rerouting, and replication of remote procedure calls. The application handles resource-related situations using techniques which may exploit application semantics, but are often independent of the system configuration, such as fault recovery procedures, handling memory and file allocation errors, and version selection for imprecise computation. In R-Shell, scheduling agents can utilize either application semantics or configuration information or both, and implement any of these features as necessary. Thus, instead of locking in the roles of the OS and the application, scheduling agents allow application designers to select the kind of behavior they want.

The salient features of the R-Shell approach are:

- **Flexible scheduling strategy:** The scheduling policies of the system can be modified easily by changing the scheduling agent functionality. For example, different programming languages can provide different scheduling agents to reflect their design philosophy. It is also easier to utilize application semantics to make more intelligent scheduling decisions.

- **Use of object-oriented model:** The R-Shell approach is truly object-oriented, in that each application object is autonomous in its scheduling decisions. Most "object-oriented" OS designs include a centralized scheduler, which makes all scheduling decisions for all objects. The behavior and correctness of every application program depends on the scheduling policies implemented in this central scheduler, and on the resource requests made by other applications to the central scheduler. This is in violation of the object-oriented philosophy, according to which each object should be an independent self-contained entity, which can be designed, implemented and verified independently. In R-Shell, since each application incorporates its own scheduling agent, which handles all situations relating to resource management, the application is more insulated from external scheduling decisions, and from other applications. This is a more object-oriented model for system design, and as we discuss later, provides portability and reusability even for real-time application software. Also, the object-oriented nature of the OS and the applications is exploited in the design of the interfaces between the different system components, and in the modeling of resources in R-Shell.
- **Fully distributed scheduling:** Since the scheduler in many object-oriented operating systems is centralized, it constitutes a performance bottleneck, as well as a single point of failure. The dependence of real-time behavior on system and network load also makes it difficult to realize many of the advantages of distributed systems, such as process migration and reconfiguration for fault-tolerance. In R-Shell, the scheduling is fully distributed, since each resource type has its own resource manager, and every application its own scheduling agent. There is no single point of failure. Also, since the scheduling agent enables applications to adapt to different situations of resource availability, it is possible to use techniques such as process migration and reconfiguration, and perhaps even to port the application to a different platform, and still obtain correct real-time behavior.

In the rest of this section, we describe some of the features of each component of the R-Shell system, and how they are used to build systems. However, before we launch into a description of the components, we first present our object-oriented hierarchical model of resources, which provides the conceptual base on which resource management in R-Shell is built. This model is used not only to represent the actual physical resources available, but also to express resource requirements and resource characteristics.

3.2 Resource modeling

One of the innovative features of R-Shell is that it models resources using an *object-oriented resource hierarchy*. This hierarchy concept allows applications to express their resource requirements more precisely. Moreover, it facilitates the innovative technique of *resource substitution* (described below) which replaces an unavailable resource with some other resource whose properties most closely match the desired properties. This technique is used in R-Shell for handling resource scarcity due to faults or preemptions.

For each type of computational resource, R-Shell describes its characteristics with an object-oriented class description. The methods of this class description correspond to the functionality provided by the resource, and the state variables capture the properties. Thus, a processor resource may have a class description which includes methods such as *execute-instruction*, *service-interrupt* etc. The state variables may include *speed* and *number-of-interrupt-levels*. Resource objects are instantiated from this class description to represent the actual physical resources in the system.

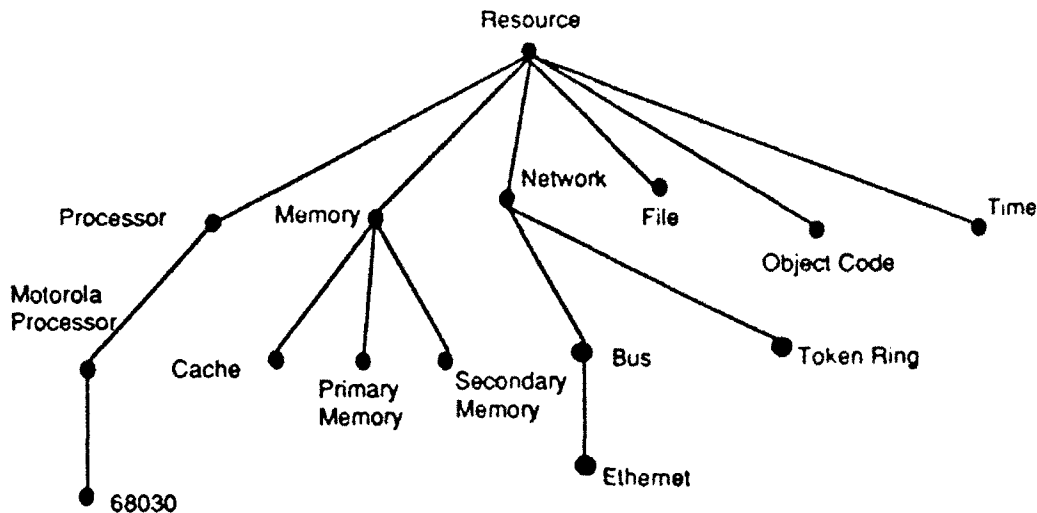


Figure 3: A (Partial) Object-Oriented Resource Hierarchy

Resource descriptions in R-Shell are organized into a resource hierarchy. Figure 3 shown an example of a partial object-oriented class hierarchy. The various resource class descriptions in the system form an object-oriented hierarchy. As we go down the hierarchy, the resource subclasses model the resource to a greater level of detail, and describe some specific subset of the resources. Thus, *Network* may have a subclass *Token-ring*, with additional methods such as *request-token* and additional properties such as *token-round-trip-time*. The purpose of the hierarchy is to capture the properties of computational resources to different levels of detail, and to distinguish between generic and specific resource types. With this, applications can express their needs for specific types of resources, by requesting resources belonging to a particular resource subclass.

The resource hierarchy is known to all components of R-Shell. The OS uses the resource hierarchy to represent physical resources, and keep track of resource allocation. Based on the resource hierarchy, the OS also knows which particular physical resource can satisfy the request for a generic resource class such as *Primary_memory*. Applications use the resource hierarchy to express their resource requirements and the assumptions they make about resource characteristics. Scheduling agents use the resource hierarchy to direct resource requests to the appropriate resource managers. The resource hierarchy also facilitates the innovative technique of resource substitution, described next.

Resource substitution

In R-Shell, applications modules (methods) express their requirements of all computational resources (actually, most of the resource requirements are derived using a schedulability analyzer, and incorporated into the scheduling agent, as described later). Moreover, using the resource hierarchy, they express the set of desired properties and characteristics, and the assumptions they make about the resource. Thus,

one application may simply need a processor which can service interrupts, whereas another may depend on the interrupt being serviced within, say, 1 microsecond. Another application may only work correctly if the processor includes a floating point accelerator. Since real-time systems typically depend on specific resource characteristics, this mechanism provides critical semantic information to the scheduling agent.

Using this semantic information, the agent can perform more intelligent resource allocation. If a processor fails, and the application must be relocated to a different processor, the agent has the information to decide which of the other processors in the system will be acceptable, particularly since the application expresses all its resource needs, such as the computational servers it requires, the assumptions it makes about communications delay to other nodes, the communication bandwidth needed etc. The agent can therefore deal with resource unavailability by substituting the scarce or unavailable resource with some other resource which is available.

One form of resource substitution occurs when a resource is replaced with another similar or equivalent resource. If an exact equivalent resource (which has all the properties desired by the application) is not available, the scheduling agent may use the information contained in the resource hierarchy to identify another resource which matches most of the properties and may still be acceptable. For example, it may provide a processor with fewer registers or without a floating point accelerator. Since all the desired properties are not met, the application may produce an approximate, *imprecise* result. Thus this form of resource substitution is similar to imprecise computation; it has the same effect of trading off result quality to deal with resource unavailability.

Another form of resource substitution occurs when the needs of one type of resource are reduced by providing more of another type of resource. Computations can often make tradeoffs between time and memory, between time and communication bandwidth, etc. Some of these tradeoffs can be built into the scheduling agent, allowing it to choose from alternative versions of library routines, such as different sorting algorithms with different resource usage. For example, the agent can increase communication bandwidth allocation to reduce communication delay, or insert message compression and decompression algorithms to reduce communications bandwidth usage at the expense of longer message delays.

R-Shell scheduling agents can support both forms of resource substitution. Resource substitution can be used for handling faults and preemptions, both at the application and the OS level.

3.3 Resource managers

The OS in R-Shell consists of a collection of resource manager objects. Each type of physical resource in the system has an OS resource manager associated with it, which schedules resources of that type. Resource managers maintain a list of resource objects, corresponding to actual resources available. They receive requests from applications for resources, and allocate resource objects to them. When an applications module makes requests for several different types of resources, the scheduling agent for the application sends requests to the resource managers for each type of resource. Resource managers interact with each other to coordinate scheduling of different applications, to avoid deadlock. When a resource manager is able to satisfy a resource request, it provides the application with a *guarantee* that the needed resources will be provided. R-Shell supports multiple levels of guarantees, including an absolute guarantee (subject to faults), a guarantee subject to preemptions by later higher-priority jobs, and a best-effort guarantee (not quite the same as no guarantee, because of exception notification, described below). Applications can select the level of guarantee they require; they can also request a lower-level guarantee if a higher-level guarantee is refused.

Upon request, resource managers can provide information about resource availability, and also accept

messages from applications specifying information about resource usage, such as preferences for certain resources. Resource managers send exception notification messages to applications if a guarantee cannot be satisfied, due to faults, or preemption of resources by higher priority tasks (in the case of the best-effort guarantee). Under these circumstances, if the resource manager cannot maintain the guarantee, it sends a message to the application (i.e. an upcall) notifying it of the *resource exception*. These messages enable applications to perform exception handling.

3.4 Support for building applications

R-Shell is designed to facilitate building object-oriented real-time applications. The support provided includes schedulability analysis tools to determine resource requirements, scheduling agents to perform resource management, and language primitives for expressing resource needs, for requesting guarantees, for performing exception handling, and for defining multiple versions of methods if imprecise computation techniques are used. Because of its widespread popularity, and the easy availability of a public domain compiler (GNU C++) which can be modified, we have chosen to add the primitives onto C++, though they can as easily be added to any other object-oriented language. The support provided by R-Shell for building real-time applications includes:

- **Expression of resource requirements:** A major emphasis in R-Shell is on complete expression of resource requirements. Current programming languages use *ad hoc* primitives to express resource needs, such as *malloc* for memory, opening files to read data, opening sockets to get access to the network etc. Also, in current systems, programs do not describe their assumptions about resources, though they do make assumptions about network speed, network bandwidth, processor speed etc. This is a major reason why programs, particularly real-time and fault-tolerant programs, are not portable to configurations other than the one they were written for.

Using the object-oriented paradigm, R-Shell allows the expression of all resource needs as requests for particular kinds of resource objects. The R-Shell programming language supports the concepts of resource objects described earlier, and the language definition includes descriptions of a standard set of resource objects, with parameters which represent their characteristics. Applications send requests to the OS, asking for allocation of instances of particular kinds of resource objects. Then they make calls (send messages) to the resource objects to perform operations on them, such as sending a message or reading some input from a file.

- **Deriving resource requirements from schedulability analysis:** The language incorporates a schedulability analyzer which uses a combination of analytical techniques and test runs to derive the resource requirements of programs, and automatically incorporate requests for resources when they are not explicitly coded in by the user. This feature relieves the application programmer of the burden of coding in explicit requests for all resources, and particularly of having to estimate execution times. The design of the analyzer is based on our previous design of the GARTAN schedulability analyzer [10]. If the method requires any special resources (such as sensors and effectors) which cannot be inferred by the analyzer, or if it makes specific assumptions about resource characteristics, then the applications programmer must insert requests for the corresponding resource class. The information derived from the schedulability analyzer is incorporated into the scheduling agent, as described in section 3.6.
- **Requests for guarantees:** The programming language also supports the notion of guarantees for method invocations. It provides the primitive *guarantee* which can be applied to a method,

which requests the OS (and the scheduling agent) to guarantee that resources will be available to guarantee the method. The *guarantee* primitive returns a value which indicates the level of guarantee provided to the method. If the guarantee is refused, the application can determine this from the return value, and take appropriate exception handling action.

- **Exception handling:** Application objects can also include methods which process exception notification messages from the operating system. If a particular type of resource becomes unavailable, the OS sends the application a message notifying it of the exception. Applications can handle these exceptions by defining appropriate exception handler methods. These exception handlers facilitate the incorporation of various fault-tolerance schemes in application objects, so that each object can encapsulate its own resource exception handling. This is in contrast to current systems, where fault detection must be done explicitly by the application by setting up and catching *signals*, or other similar mechanisms, and the language may provide little or no support for performing fault-tolerance.
- **Version selection for imprecise computation:** If the application chooses to use the techniques of imprecise computation for handling resource scarcity and faults, then the programming language provides support for defining multiple versions, and for performing version selection at run-time. Application objects may define several different implementations of any method, using the techniques of imprecise computation. These different versions differ in their resource requirements, and produce different qualities of result. At run-time, that version is selected for execution whose result quality is the best, from among those whose resource requirements can be guaranteed. This version selection can be performed by the scheduling agent for the application.

It should be noted that the concept of scheduling agents is language-independent, and that scheduling agents can be provided even for languages which do not provide the special support facilities described here. These particular facilities are provided in R-Shell because we believe they are very useful for building real-time applications, however other languages and OS designs may prefer to provide different functionality in scheduling agents, and different language and run-time support facilities.

3.5 Run-time support: Scheduling agents

Every application has a scheduling agent associated with it at run-time. This scheduling agent is part of the run-time support environment for the application, and is an enhancement of the conventional concept of the RTE. Whereas conventional RTEs provide functions such as object management and resolution of method invocations, the scheduling agent can generalize this to include scheduling functions such as resource management and version selection.

The application dictates the functionality to be provided by the scheduling agent. For example, if the application uses the *guarantee* primitive, then the scheduling agent will perform resource guarantee functions. Similarly, if the application defines several different versions of methods, then the scheduling agent will include version selection functionality. Applications designers can also specify that certain functionality should or should not be included in the scheduling agent, e.g. whether or not the scheduling agent should perform resource substitution.

The functionality provided by scheduling agents will also depend on the programming language. For example, a scheduling agent for Ada would include support for those **PRAGMAS** which are implemented on the particular system. The implementation of the scheduling agent would depend on the capabilities

provided by the underlying OS. Thus, resource substitution functionality can only be provided if the OS supports the resource model.

Figure 4 illustrates the operation of a typical scheduling agent for a real-time application which requires guarantees, and uses imprecise computation. The application expresses resource requirements for each version of each method. When the application requests that a method invocation be guaranteed, the scheduling agent is activated. The agent interacts with the OS resource managers to determine whether all the resources needed by the application are available. Based on this information, it selects a version of the method to execute, and requests each resource manager involved to guarantee its resource requirements. By determining resource availability in advance, the agent performs the coordination function of avoiding situations where some resources are guaranteed, but others are refused. If a guarantee is provided, but is subsequently violated (due to faults or preemptions), then the agent itself attempts to handle the exception situation by using resource substitution or selecting an alternative version to execute. In the event that it is unsuccessful, or if guarantees were refused in the first place and no alternative version is available, the agent notifies the application and transfers control to the appropriate exception handler. If the application does not provide exception handlers, scheduling agents themselves provide default exception handlers (perhaps simply print an error message and quit).

Thus scheduling agents can provide resource management functionality for the application, without adding unnecessary overhead to those applications which do not require the functionality. It may be argued that scheduling agents unnecessarily duplicate in a higher layer some functions which can be provided in the OS itself, and therefore add processing and memory overhead. However, this possible disadvantage is more than offset by the elimination of the unused functionality which is inevitable when using a sophisticated OS, and by the use of application semantic information. For example, if the OS scheduler supported imprecise computation, and were to perform version selection, it may spend time in sorting the various versions in order of decreasing resource requirements and increasing result quality, so that it may first try to guarantee the better versions, and if they fail, then try poorer versions. On the other hand, the scheduling agent can have the version order hard-coded into itself, and thus avoid the sorting step altogether. We can also avoid replication of the same scheduling agent code in different applications by making scheduling agent code re-entrant, and sharing the code among different applications.

3.6 Construction of scheduling agents

Scheduling agents are generated during compilation and in the post-compilation phase. The following information is used in the generation of scheduling agents:

- Resource requirements information expressed in the application.
- Resource requirements derived from schedulability analysis.
- Application semantic information expressed using the language constructs, such as: multiple versions of methods, **guarantee** primitives, **PRAGMAs**.
- System configuration information, as obtained from the OS at agent generation time.
- Library routines which implement special resource management techniques, such as various fault-tolerance techniques, version selection and resource substitution.
- Exception handlers defined in the application.

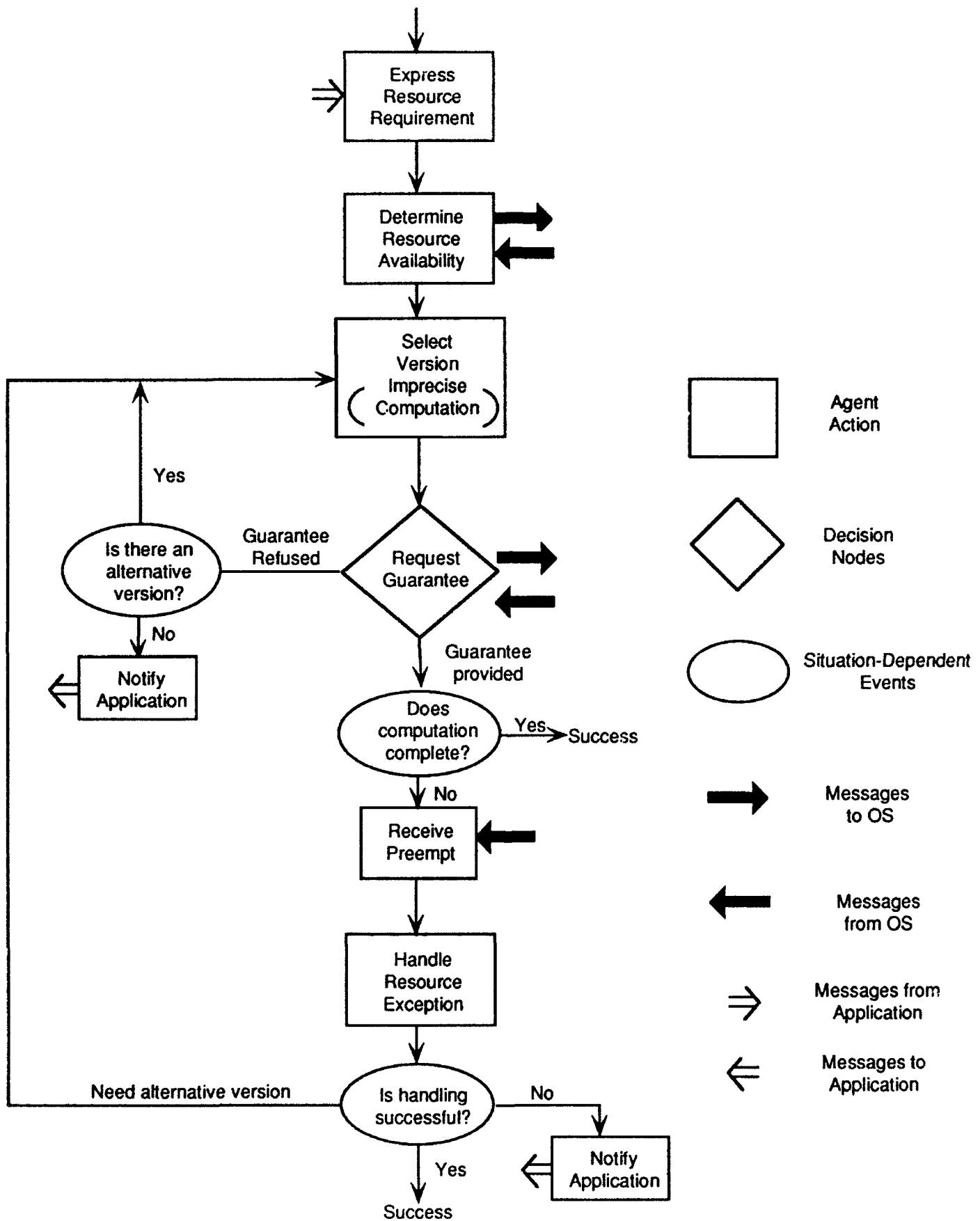


Figure 4: Example of Scheduling Agent Operation

- Default resource exception handling routines.
- Directives from the application programmer specifying functionality to be provided by the scheduling agent.

The scheduling agent combines the information about resource requirements obtained from the application and from schedulability analysis, to derive the resource requirements of each application method. Using the resource model, the resource requirements are translated into guarantee requests to individual OS resource managers. The use of the guarantee primitive triggers the execution of this resource request code in the scheduling agent. If the application defines multiple versions, the agent performs version selection before requesting guarantees. Receiving exception notification from the OS triggers the execution of special routines such as fault handling and resource substitution. The inclusion of these special routines, and the strategies for obtaining guarantees, performing version selection etc. are all dependent on the directives issued by the application programmer. When a fault notification is received, the scheduling agent first attempts to execute any applicable special routines which it may contain. If there are no applicable special routines, then the agent triggers the execution of any fault handlers defined by the application for the situation. If there are none, then it executes the default handler.

It must be emphasized that scheduling agents can be built for any programming language and any operating system, to perform any functions desired by the application, as long as the run-time overhead of providing that functionality is acceptable to the application. In R-Shell, we propose the particular design and functions outlined here, because we believe that these are particularly appropriate for building object-oriented, distributed, fault-tolerant real-time applications which can adapt to a variety of dynamic resource requirements and resource availability. We believe that the performance of this approach will be comparable to, if not better than, that of systems which provide equivalent functionality by incorporating features in the OS.

4 Conclusion

The primary contribution of R-Shell is its use of scheduling agents to integrate the resource management functionality that is conventionally provided separately by the OS and the application. Since scheduling agents have access to system configuration information as well as application semantic information, they can implement sophisticated resource management strategies. By avoiding the inclusion of unnecessary functionality in applications which do not require it, scheduling agents can reduce system complexity and improve performance. Scheduling agents provide a flexible, fully distributed, object-oriented solution to the problem of resource management in real-time systems.

In addition to scheduling agents, R-Shell also includes some other innovative features: the use of a resource hierarchy to model resource characteristics, enabling applications to express resource requirements better; explicit interaction between applications and OS to obtain guarantees and information about resource availability; the use of exception notification and exception handlers for fault detection and fault-tolerance; and the technique of resource substitution for dealing with resource scarcity.

5 Bibliography

References

- [1] R. H. Campbell, G. Johnston and V. Russo, "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)", *ACM Operating Systems Review*, vol. 21, no. 3, pp. 9-17, July 1987.
- [2] P. Gopinath and K. Schwan, "CHAOS: Why one cannot have only an operating system for real-time applications", *Operating Systems Review*, vol. 23, No. 3, pp. 106-125, July 1989.
- [3] C. M. Krishna and Y. H. Lee (guest editors), "Special issue on real-time systems" *IEEE Computer*, vol. 24, No. 5, May 1991.
- [4] Shem-Tov Levi, S. K. Tripathi, S. Carson and A. Agarwala, "The MARUTI hard real-time operating system", *Operating Systems Review*, vol. 23, No. 3, pp. 90-105, July 1989.
- [5] K. J. Lin, S. Natarajan and J. W. S. Liu, "Imprecise Results: Utilizing Partial Computations in Real-Time Systems" *Proc. I.E.E.E. Real-Time Systems Symposium* 1987, pp. 210-217.
- [6] C. D. Marlin, W. Zhao, G. Doherty and A. Bohonis, "GARTL: A Real-time Programming Language Based on Multi-version Computation", *Proc. I.E.E.E. Int. Conf. on Computer Languages* 1990, pp. 107-115.
- [7] S. Natarajan and K. J. Lin "FLEX: Towards Flexible Real-Time Programs" *Computer Languages* vol. 16, no.1, pp.65-79, 1991.
- [8] S. Natarajan and W. Zhao, "Building Dynamic Real-Time Systems", *IEEE Software* special issue on Real-Time Systems Development, Sep. 1992, to appear.
- [9] B. Randell, "System structure for software fault tolerance", *IEEE Trans. Software Eng.*, vol. SE-1, pp.220-232, June 1975.
- [10] K. J. Ransom, C. D. Marlin and W. Zhao, "GARTEN: A programming environment for real-time software development" *Proc. 8th Workshop on Real-Time Software and Operating Systems*, Atlanta, GA, May 1991.
- [11] J. Stankovic, "Misconceptions about real-time computing", *IEEE Computer*, vol. 21, No. 10, Oct. 1988.
- [12] J. Stankovic and K. Ramamritham, "The Spring kernel: A new paradigm for real-time operating systems", *Operating Systems Review*, vol. 23, No. 3, pp. 54-71, July 1989.
- [13] H. Tokuda and C. W. Mercer, "ARTS: A distributed real-time kernel", *Operating Systems Review*, vol. 23, No. 3, pp. 29-53, July 1989.
- [14] W. Zhao, K. Ramamritham and J. Stankovic, "Pre-emptive scheduling under time and resource constraints", *IEEE Transactions on Computers*, vol. 36, No.8, Aug. 1987.
- [15] W. Zhao (guest editor), "Special issue on real-time operating systems", *Operating Systems Review*, vol. 23, No. 3, July 1989.

FOUNDATIONS OF SYSTEM ENGINEERING

An Extended Event Descriptor for Real-Time Systems

11 June, 1992

CHRISTOPHER BLOW AND DIETER ROMBACH
University of Maryland

Abstract¹—Event based state transitions are central to some techniques for specification of real time systems. As a part of the Software Cost Reduction (SCR) project at the Naval Research Laboratory, an event descriptor notation was defined in which events are described in terms of changes to boolean predicates. Follow-on research produced other models and definitions of the SCR event descriptor. However, all of these definitions have limitations in terms of their ability to describe certain useful classes of events. We therefore establish a rationale for evaluating event descriptors in terms of generality, implementability, and verifiability. To fulfill these requirements, we propose an extended event descriptor which allows expression of a larger class of events. The new descriptor has the added advantage of displaying related functional and timing specifications together, which allows easier understanding of the meaning of an event. We explore the meaning of the event descriptor, both in terms of a formal definition and the code which is required to implement the specification.

1. Introduction

In software specification documents for real time systems, functions to be incorporated in the software may be specified such that their output values change in response to external events. Such functions are most easily specified in terms of state transitions, where the transitions are triggered by events. An example would be a function which illuminates a warning light whenever a fluid level exceeds a certain amount and which then extinguishes the light if an acknowledgment button is depressed by the operator. This paper concerns attempts to produce a notation to define and describe such events in software specifications.

¹This work was supported, in part, by the National Science Foundation, under grant CCR-9057874 to the University of Maryland.

Under the direction of D. Parnas, the Naval Research Laboratory's Software Cost Reduction (SCR) project provided an important contribution to formal requirements specification of real-time software by producing a formal specification method which was validated by application to a large software system: the already existing A-7 aircraft Operational Flight Program. That specification, in its final form, ran to approximately 500 pages [NRL 88]. As a part of that specification method, SCR developed a notation to describe events, using an extension of first order logic [HENI 80]. In this notation, an atomic event is described by the notation "*@T(condition)*," where *condition* is a boolean predicate. The event described by the *At-True* expression is considered to occur at any instant in time when the predicate *condition* transitions from false to true. "*@F*" indicates the converse. The notation was further extended to incorporate a guarding condition, in the form "*@T(condition1) WHILE (condition2)*". This event occurs at the instant when *condition1* transitions from false to true, given that *condition2* is true at the same time.

The description of the *@T* notation in the previous paragraph is strictly intuitive. It uses the words "instant" and "at the same time." However, since computer software does not deal in infinitesimal instants or simultaneity, no program can monitor both *event* and *condition* at the same time. If the notation is to specify the behavior of computer software, then its formal definition must be unambiguous and implementable with software, and the implementation must be verifiable.

We propose three main aspects by which an event descriptor should be evaluated. First, it should be general enough to cover a class of useful, reasonable events. We will explore certain types of events that require a more expressive descriptor. Second, it must be unambiguous and should allow implementation in code in a straightforward

manner. Thus, it should not require that unduly complex code be written to detect simple events. Third, the descriptor should enable the verification of the code that is written to implement it. There should be a practical means to examine the code to determine that it does in fact fulfill the specification. These can be difficult standards to fulfill. As we will demonstrate, event detection in code is often a simple matter, perhaps requiring no more than a single condition statement. Yet where complex definitions of events are required, the code that implements those events may become quite intricate.

In section 2, we address the main topic of this paper, which is the issue of generality, or the variety of occurrences that can be covered by the definition of *event*. Section 3 covers previous work that has been published concerning the definition of the @T event descriptor. In Section 4, we motivate the selection of an improved event descriptor with a detailed description of the requirements that the descriptor should fulfill. Our proposed descriptor is presented in Section 5 and then defined and implemented in code in Section 6. Section 7 covers other factors involved with implementation, concerning the nature of inputs to the program. In Section 8, we evaluate our descriptor and the previous work in terms of the proposed requirements.

2. Generality in Event Description

2.1 Event Sequencing and Timing

Since single processor systems can test only a single external condition at a time, and communication and resource contention prevent distributed systems from comparing external data values at the same time, the definition should imply a specific sequence of tests upon external variables. It is this need for a sequence of testing that gives rise to the most fundamental ambiguity in the "@T(A) WHILE (B)" notation: should the *condition* be tested *before* the *event*, *after* it, *both* before and after, or *either* before or after? In practice, it is useful to be able to specify any of the four situations. The following paragraphs provide examples in which each of the four would be appropriate.

Before. Suppose that the occurrence of the event @T(A) may make the value of B inaccessible. For example, if A is an explosion, and B is a reading of ambient temperature from a delicate device adjacent to the explosive, B may be invalid once A becomes true. In this case, the specification writer ("specifier") should be able to dictate that, when A transitions from false to true, the WHILE condition should use the value of B obtained most recently before the event @T(A). The same order would also apply if @T(A) WHEN (B) initiates a process, @F(B) terminates the process, and it is essential that the process be initiated in cases where the same external occurrence may cause @F(B) shortly after @T(A).

After. Suppose B is valid only after @T(A) occurs. For example, if B were the output of a peak-reading pressure meter that measured the intensity of the explosion in the previous example, it would be essential to specify that B be checked only after the event @T(A). Or, in the second example of part the previous paragraph, if there is a safety constraint which dictates that the process *not* be initiated in cases where @T(A) and @F(B) are caused by the same external occurrence, the specifier may wish to dictate that the value of B be checked after @T(A).

Both. Suppose there are two types of events that may cause @T(A). One type will affect only A; the other type will make A true and will reverse the value of B. If only the first type of event should trigger the function, then the value of B must be checked both before and after the value of A.

Either. In discussing difficult and complex cases, we must remember that, in most cases, the sequencing and timing of event detection will not be critical. Using the explosive-temperature example, if the thermometer were placed far enough from the explosion not be affected, the sensing of the explosion and checking of the temperature could occur in any sequence. The specification might only require that the temperature value be sampled at a time reasonably close to the explosion (i.e. not an hour prior).

These examples may also be extended to cover cases where indeterminacy in data-arrival timing dictates that the value of B be examined within a certain interval prior to, or after, the occurrence of @T(A). For example, in the measurement of the

explosive's ambient temperature, we may require that B be measured no earlier than one minute prior to $@T(A)$, but no later than one millisecond prior to the event. A good event descriptor should be able to express any of these situations.

2.2 Input Behavior

Another consideration is the ability of the specification to handle ill-behaved input signals. Three examples of such difficult behavior that must be addressed are shown in Figure 1. In Case 1, a digital signal is produced by a mechanical switch which suffers from "bouncing" as it is turned on. The signal oscillates for a time, before settling into its new state. The event $@T(\text{Switch} = \text{On})$ would then be detected four times, for a single movement of the switch. In Case 2, an inherently analog signal, temperature, is being measured with an accuracy (± 15) worse than the precision of the digital input (± 0.05). Thus, the digital signal displays random oscillation about its true value (indicated by the dotted line). If the event of interest were $@T(\text{Temp} \geq 98.4)$, this oscillation could cause the event to be detected multiple times during a period when the temperature was unchanging. $@T(\text{Temp} \geq 98.7)$ would be detected twice, even though the true value was constantly increasing. In Case 3, the temperature is being measured at a better level of accuracy (± 0.01) than the digital precision (± 0.05). The signal therefore tends to oscillate about the true value. A temperature of 98.42 results in an oscillating digital value that spends 60% of the time at 98.40 and 40% at 98.45. Even with a constantly increasing temperature, the digital input may

show brief decreases. In the figure, such an oscillation would cause the event $@T(\text{Temp} \geq 98.6)$ to be detected twice.

Thus, in any of these three cases, an $@T$ expression might be triggered many times due to an oscillating or "bouncing" value. If we are to allow for system designs that require the above events to only occur only once in each such case, we will need to extend the descriptor to provide for more complete handling of the inputs.

3. Previous work

The SCR project [NRL 88] formally defined the At-True notation as follows:

This notation calls for the evaluation of a condition simultaneously with the detection of an event. Since this is not possible in any real implementation, we define the meaning of an expression such as $@T(X)$ WHEN (Y) thus: Let t_s be the time at which $@T(X)$ is detected by the software. Then there is a time interval ϵ such that if Y is true (false) for all of $|t_s - \epsilon, t_s|$, or for all of $|t_s, t_s + \epsilon|$, or for all of $|t_s - \epsilon, t_s + \epsilon|$, then the expression is (is not) considered to be satisfied. The behavior of the system when Y is true for only part of the interval is not defined; it may or may not behave as though the expression were satisfied. Which interval we use as the requirement, as well as the value of ϵ , is defined in the Accuracy chapter [of the SCR specifications] for each such event.

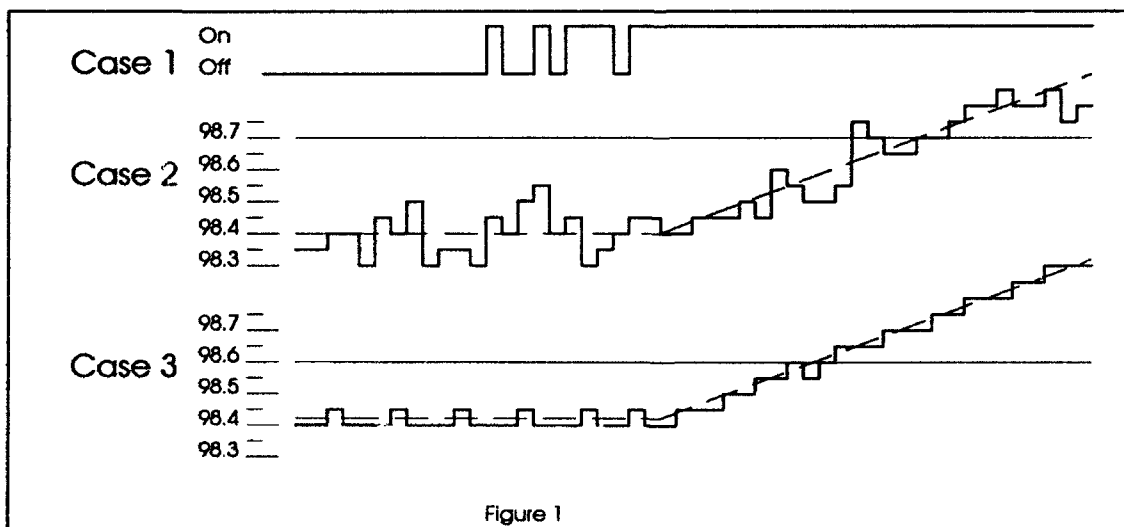


Figure 1

This definition acknowledges the impossibility of simultaneous evaluation of conditions and events and permits the specifier to choose whether the condition should be true before, after, or before and after, the software detects the event. The specifier may also choose the interval (epsilon) over which the condition must be true. However, this definition has several weaknesses. Most importantly, it fails to directly relate the behavior of the event variable (X) to the WHEN variable (Y). Y's behavior is related to the time when the event is detected by the software, rather than the time when X's value changes. The delay that is allowable in detecting the event and the rate at which the event variable must be checked are not directly specified at all in SCR. Only where the event triggers the performance of a function is a "Maximum Delay to Completion" given for that function. The placement of the timing constraints in a separate chapter makes it difficult to understand the correct meaning of an event. There is no provision for avoiding spurious events resulting from bouncing or oscillating inputs. The definition does not make any attempt to deal with situations where the WHILE condition may be expected to change shortly before or after the event. No provision is made to specify that the condition be checked both before and after the event.

[FAUL 89] provides the specification method of SCR with both a theoretical basis, in finite state automata, and a practical means of translation into code, using State Transition Event (STE) synchronization of cooperating sequential processes. This mechanism is intended to supply the temporal rigor required of Hard Real Time embedded systems and to address the often complex issue of process scheduling that can arise in an SCR specification. This dissertation makes a very important point on the ordering of events. Whether due to the sequential nature of single processors, or communication delays in multiple processor architectures, there is always an indeterminacy in the detection of events. Only a partial ordering of events is available in a computer. Therefore, his definition of finite state automata is extended to incorporate a relation describing near-simultaneity. However, the specific definition of the @T construction is not addressed.

[SCHO 90] also contributes to the theoretical underpinning for the SCR method. *Event classes* are

defined as instants in time when a predicate P on an environmental state function s and time t is true $[P(s,t)]$. A mechanism is provided for expressing the ideal behavior of a system, with acceptable deviations (in time and precision) from the ideal. A grammar is provided for the generation of "event class forms," including At-True expressions. The following definition of "@T(P_1) WHEN (P_2)" is given:

$$EC(\exists \epsilon[(\epsilon > 0) \wedge \forall \delta[(0 < \delta \leq \epsilon) \rightarrow (\neg P_1(s, t - \delta) \wedge P_2(s, t - \delta) \wedge P_1(s, t))]])$$

where EC is the Event Class consisting of all times t where the expression evaluates to true. In other words, P_1 must have been false, and P_2 true for all of some interval before P_1 became true. Unlike SCR, this definition does precisely state which behavior of both variables constitutes an event. However, only intervals before the event are supported. In other areas, this definition suffers from the same lack of flexibility as the SCR descriptor.

4. Rationale for Choosing a Definition of @T

As mentioned in the introduction, we believe that an event descriptor should achieve the objectives of generality, implementability, and verifiability. We now explore these criteria in detail. In section 2, we demonstrated the need to express limitations on events concerning the *sequencing* of inputs and *bouncing or oscillating* inputs. We contend that, to fulfill the criterion of generality, an event descriptor should be able to specify input sequencing whenever it is desired. There should also be provision for useful specifications dealing with the ill-behaved inputs presented in section 2.2.

Regardless of the intuitive and theoretical definitions of the At-True notation, if it is to be used in the specification of software, then its value lies in the possibility of implementation in code. Thus, the definition of the notation should be appropriate to the goal of producing code, not to any abstract sense of neatness or symmetry. It should completely describe the circumstances under which the code *must*, *may*, and *may not* detect an event. Furthermore, the code implied by the use of the notation should not be any more complex than the problem requires. In most cases, even in hard real time systems, only a minimum polling rate for

the inputs needs to be specified, and exact timing and sequencing of the polling is not a specification issue. Yet, in those instances where the relevant external variables are not independent, or the inputs are ill-behaved, the specifier should be able to express more exact requirements. Some of the additional complexity involved in epsilon intervals can be justified in the case of hard real time systems, where temporal precision is crucial. However, the "hard real time" qualifier means only that the required functions must always be completed in the specified time. It does not necessarily imply that the drawing of distinctions between event ordering need be any more or less precise than in non- hard real time systems.

In accordance with this reasoning, we maintain that an event descriptor fulfills the requirement of implementability if it completely defines the circumstances when events will be detected. In its least complex form, the descriptor should be implemented by code that performs no more than a straightforward check of the input values. It should also be practical to implement the detection of more complex event descriptions. Further, the definition must be applicable for cases in which any atomic predicate of the At-True notation is either polled or interrupt driven. In this paper, "interrupt driven" refers only to those input data items that result in immediate suspension of code execution and branching to separate code (possibly pre-empted by higher priority interrupts). Note that this definition of "interrupt driven" does not include data items, such as buffered keyboard inputs, that cause system-level interrupts without changing the order of execution of statements in

the software being specified. At the level of the software being specified, the data item is interrupt-driven only if the software itself "hooks" the interrupt.

Finally, we propose the goal of verifiability. Once the code is written to implement an event descriptor, it should be possible to verify the correctness of that implementation by identifying the code elements that fulfill the functional and temporal requirements of the definition. We shall provide an example of such a verification at the end of the next section.

To depict the fulfillment of these criteria, we shall use the tabular form as shown at the bottom of the page.

As previously noted, the original SCR descriptor definition provides for all of the variable-checking sequences except "both." The Event Class definition allows only the "before" sequence. Neither of these definitions provide the means to handle ill-behaved inputs or time delays before and after the event. The original definition is incomplete, as it does not address when the event variable itself should be checked. Event Classes correct this omission. Both of these definitions allow simple code to be generated to detect events. Since they do not address complex input behavior and polling and interrupts, they are not rated in these areas. Due to its vague description of event detection, and its relegation of timing constraints to a separate chapter, the SCR descriptor does not fulfill the verification criterion. It is unclear whether Event Classes would improve upon this.

	Original SCR [NRL 88]	Event Classes [SCHO 90]
Generality:		
Sequencing of WHEN condition-		
Before event	Y	Y
After event	Y	N
Both	N	N
Either	Y	N
Time delays	Y	N
Input Behavior-		
Bouncing	N	N
Oscillating	N	N
Implementation:		
Complete definition of event	N	Y
Simple form	Y	Y
Complex form		
Polled / Interrupt variables		
Verification:	N	~

5. The Extended Event Descriptor

5.1 Extensions

The requirement of generality will be provided for by a series of extensions of the @T notation. By subscripting the @T and WHEN symbols with numbers and symbols representing temporal limits, the exact nature of the timing constraints can be specified. The most elementary form of our new descriptor consists of the basic SCR format with a time subscript added to the event: $@T_x(A)$ WHEN (B) . This form requires that $@T(A)$ be detected if A remains false for at least x units of time and then remains true for at least x units of time. The value of B may be checked at any time within x units of time before or after the time when A changes value. Assuming that both values are polled, any scheme that polls A at intervals of no more than x would be acceptable. B may either be polled regularly at intervals of no more than x , or it may be polled immediately after $@T(A)$ is detected. This allows the programmer maximum flexibility in implementing the specification. This form of the descriptor is equivalent in definition to the *event classes* of [SCHO 90], with the epsilon interval applying to both sides of the event.

To allow for the "bouncing" of inputs, we provide our next extension to the notation. The subscript after the @T portion may consist of two comma separated elements, separately describing the times that the expression must remain false, and then true. For example, $@T_{x,y}(A)$ WHEN (B) would indicate that the event must be detected whenever A is false for at least x time and then true for at least y time. The bouncing mechanical switch can be accommodated by this notation, where x and y may be set to the minimum amount of time that the switch will remain in each state before changing. So long as the "settling" time for the switch's bouncing is significantly less than x and y , this will ensure that only one event is generated for each movement of the switch.

When the inputs oscillate, as in cases 2 and 3 of Figure 1, the @T notation must be further extended by providing a pre-condition for the event. $@T_{x1,x2}(A' \Rightarrow A)$ WHEN (B) indicates that the event must be detected whenever the precondition A' is true for at least $x1$ time, and then A is true for at least $x2$ time. For the example of Case 2 of figure 1, if we wanted to specify an

event in which temperature increases past 98.7 degrees, we could ensure that spurious events would not be detected by specifying $@T_{2,1}([Temp \leq 98.4] \Rightarrow [Temp > 98.7])$. This event must be detected any time the temperature stays at or below 98.4 for at least 2 units of time and subsequently increases past 98.7 for at least 1 unit of time. It must not be triggered until after the temperature is found to be less than or equal to 98.4 for at least one sample. Similarly, the requirement of Case 3 may be satisfied by $@T_{1,1}([Temp \leq 98.5] \Rightarrow [Temp \geq 98.6])$.

The next extension for the notation allows the specification of the sampling order of the WHEN clause. A subscript after the WHEN clause can contain a relational sign ($<$ or $>$) and, optionally, a number representing a time delay. In understanding this use of the relational signs, keep in mind that they are used to delineate the sampling order of the clauses. Thus, $@T_x(A)$ WHEN $_{<}(B)$ requires that B be sampled after the event $@T(A)$ has occurred. $@T_x(A)$ WHEN $_{>y}(B)$ indicates that B should be sampled at least y units of time after $@T(A)$. In both cases, B should be sampled no later than x units of time after the event. Multiple WHEN clauses may be used if different sampling orders are required for different conditions.

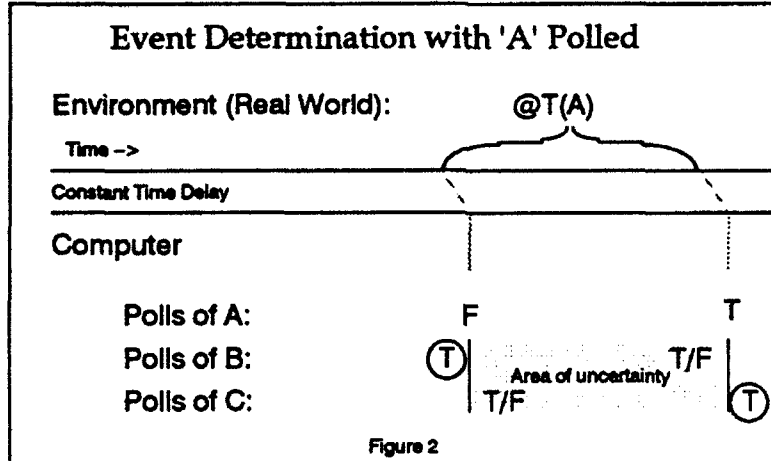
5.2. Polling and Interrupts

This completes our extension of the event descriptor. A formal definition, along with coding examples, follows. First, however, another dimension of the sampling issue must be addressed to fulfill the goal of implementability—that of the interaction between input type (polled or interrupt) and sampling sequence. To illustrate this interaction, we will investigate a series of examples based upon an event specified by $@T_x(A)$ WHEN $_{>}(B)$ WHEN $_{<}(C)$. This specification dictates that B must be sampled before the event, and C after.

If A , B , and C are polled variables, this specification is most easily implemented by conducting a poll of all three variables every x units of time, using the order $B-A-C$. The event $@T_x(A)$ is detected whenever successive polls of A result in values of false and then true. Since the exact timing of the external event cannot be determined within the interval between the two polls of A , the value used for B must be checked before the first

poll of A (and retained for a full polling cycle), and C must be checked after the last poll of A. If A is polled, but B and C are interrupt driven, the same effect is achieved by storing the value of B for one polling cycle, and sampling C after polling A.

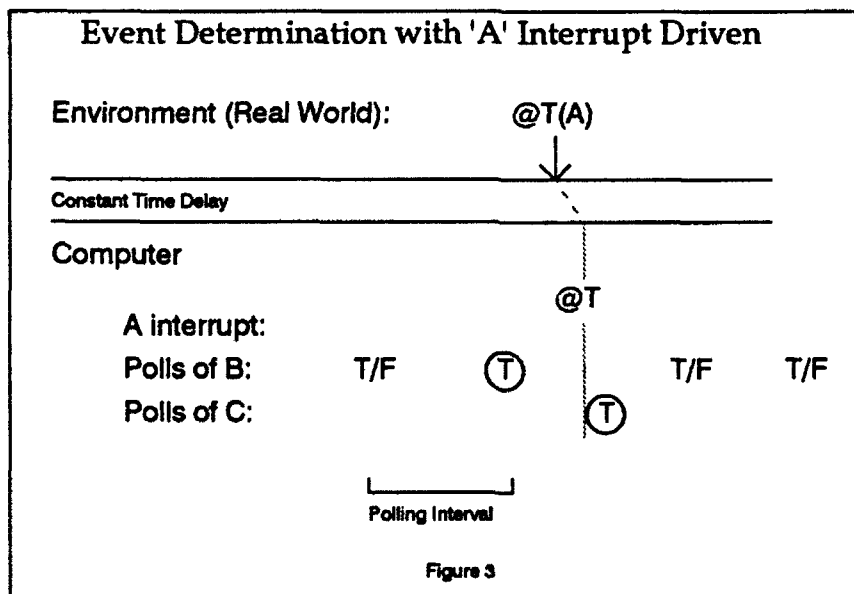
If A is interrupt driven, as illustrated in Figure 3, the timing of the event @T(A) will be better known. Where B and C are polled values, the last poll of B prior to the event will be used, and C will be polled as part of the interrupt routine. If B is interrupt driven, A must have a higher

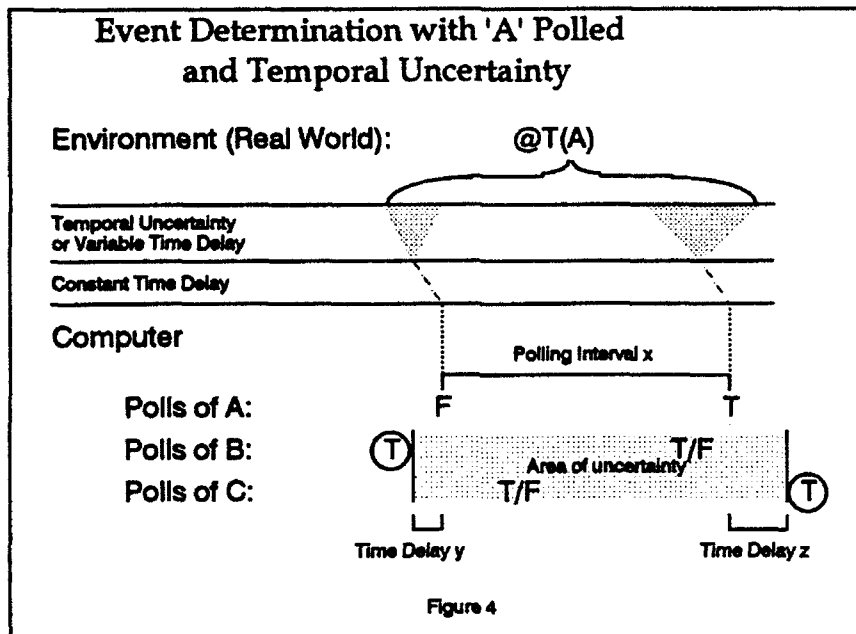


This is illustrated in Figure 2. As time elapses (from left to right), the software conducts regular polls of B, A, and C, in that order. The successive values of false and true for A indicate that the event @T(A) occurred sometime within the shaded zone. Therefore, the values of B and C in the shaded zone, whether true or false (indicated in the figure by "T/F"), are not useful; the circled values must be used. Although there is a time delay inherent in the software's access to external data, in this example that delay is relatively constant for all data.

interrupt priority, to prevent changes to the previous value of B during A's interrupt handling routine. Conversely, C must have a higher interrupt priority than A, since A's interrupt-handling routine must be able to receive an updated value of C.

The above reasoning is all that is needed in cases where the required intervals between the event and the sampling of B and C are less than a processor instruction cycle. Where that is not the case, either due to temporal indeterminacy or external relations between the conditions, delays



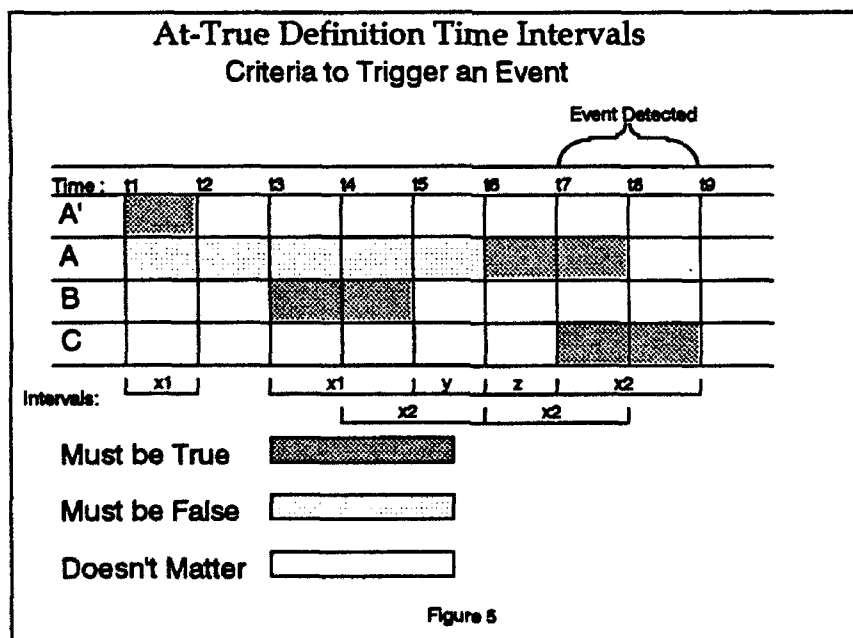


must be inserted into the sequence of variable value checks, using numeric values with the WHEN expressions. Figure 4 illustrates the specification $@T_x(A) \text{ WHEN}_{\rightarrow y}(B) \text{ WHEN}_{\leftarrow z}(C)$.

In Figure 4 the time intervals, representing required delays and/or temporal uncertainties, have further spread the time interval over which the real-world event may be considered to have occurred, relative to the B and C conditions. The polling of B and C must be separated from the polling of A by at least the amount of the delay values.

6. Definition and Implementation

We have asserted that the event notation should be defined in terms of the code that it requires. Therefore, for a given event expressed in our notation, we define when the code *must*, *may*, and *must not* detect an event, based upon the information available to the software system. Predicates will be expressed with a time parameter; $P(t)$ is true iff a value polled (or read from memory, if interrupt driven) at time t makes P true. This definition is described in terms of time intervals, delineated by nine points in time, as illustrated in Figure 5.



DEFINITION

The fully extended event notation $@T_{x1,x2}(A' \Rightarrow A)$ WHEN $_{\geq y}$ (B) WHEN $_{\leq z}$ (C), where $x1 > 0 \wedge x2 > 0 \wedge y \geq 0 \wedge z \geq 0 \wedge y < x2 \wedge z < x2 \wedge [(A' \wedge A) \rightarrow \text{false}]$, is defined as follows:

PART 1

the code must detect an event exactly once for any time interval (t7,t9) where:

$\exists(t1,t2,t3,t4,t5,t6,t8)$ such that $t1+x1 \leq t2 \wedge t2 \leq t6 \wedge t3+x1 \leq t5 \wedge t4+x2 \leq t6 \wedge t5+y \leq t6 \wedge t6+z \leq t7 \wedge t6+x2 \leq t8 \wedge t7+x2 \leq t9$ and

$\forall t[(t1 \leq t < t2) \rightarrow A'(t)]$ and
{precondition satisfied}
 $\forall t[(t2 \leq t < t6) \rightarrow \neg A(t)]$ and
{A is false after precondition but before t6}
 $\forall t[(t4 \leq t < t6) \rightarrow \neg A(t)]$ and
{A is false for at least interval x2}
 $\forall t[(t3 \leq t < t5) \rightarrow B(t)]$ and
{B is true for at least interval x1}
 $\forall t[(t5 \leq t < t6) \rightarrow \neg A(t)]$ and
{A is false during interval y before t6}
 $\forall t[(t6 \leq t < t8) \rightarrow A(t)]$ and
{A is true for at least interval x2 after t6}
 $\forall t[(t7 < t \leq t9) \rightarrow C(t)]$.
{C is true for at least interval x2}

PART 2

The code may detect an event during any interval (t7,t9) where

$\exists t[(t1 \leq t < t2) \rightarrow A'(t)]$ and
{precondition satisfied at least momentarily}
 $\forall t',t''\{[t1 \leq t' < t'' \leq t2 \wedge t''-t' \geq x2] \rightarrow$
 $\exists t[(t' \leq t < t'') \rightarrow A'(t)]\}$ and
{precondition satisfied at least once during every interval of length x2 between t1 and t2}
 $\exists t[(t1 \leq t < t6) \rightarrow \neg A(t)]$ and
{A is false at least momentarily before t6}
 $\exists t[(t3 \leq t < t5) \rightarrow B(t)]$ and
{B is true at least momentarily}
 $\exists t[(t6 \leq t < t8) \rightarrow A(t)]$ and
{A is true at least momentarily during interval x2 after t6}
 $\exists t[(t7 < t \leq t9) \rightarrow C(t)]$ and
{C is true at least momentarily during interval x2}

PART 1 of the definition ensures that each condition will be true throughout the appropriate time interval. So long as the criteria depicted in

the figure and in PART 1 occur, the code *must* detect the event once. That detection can occur no earlier than time t7 and should be no later than shortly after t9. PART 2 ensures that each condition is true at least once during each interval during which the code should check the value of that condition. If so, the code *may* detect the event. This is equivalent to the circumstances in which the original SCR descriptor was "undefined." If neither PART 1 nor PART 2 are true, the code must not detect the event. Note that, if the precondition interval (x1) is longer than the basic polling interval (x2), then PART 2 allows detection of the event only if the precondition is true at least once during every interval of length x2 between t1 and t2. This has the effect of requiring that the code check the precondition as often as it checks the other conditions. Note that the time interval from t1 to t2 must occur before t6, but no ordering is required between these times and t3 and t4. In other words, the precondition is satisfied if its expression is true for an interval of x1 at any time prior to A becoming true.

The following example is a Pascal program that implements the expression $@T_{x1,x2}(A' \Rightarrow A)$ WHEN $_{\geq y}$ (B) WHEN $_{\leq z}$ (C), using polling of A, B, and C. For reasons of brevity, this code is based upon the assumptions that $y+z \ll x2$ and that $x1 = x2$. If these assumptions were not justified, the code would become more complex. An extension of this code, relaxing the assumption to allow $x1 \geq x2$, is included as an appendix, in Section 11 of this paper.

```

001 program RealTimeProg;
002 var
003   A, APrime, B, C : Boolean;
004   TimeNow, NextTime, X1, X2, Y, Z : TimeType;
005 const
006   SmallTime : TimeType = {SmallValue};           {See explanation below}
007
008 {functions Time, Initialize, Delay, PollA, PollB, PollC, and PollAPrime
009 and procedure ReportEventDetected not shown}
010
011 function AtTrueABC : Boolean;           {Returns true if event occurs}
012 var
013   PrevA, PrevB : Boolean;
014 begin
015   PrevA := A;                           {Save previous values of A & B}
016   PrevB := B;
017   PollB(B);                             {Poll A, B, C with delays of Y and Z}
018   Delay(Y);
019   if not APrime then PollAPrime(APrime); {If APrime true, don't poll}
020   PollA(A);
021   Delay(Z);
022   PollC(C);
023   AtTrueABC := APrime and not PrevA and A and PrevB and C;
024   if A then APrime := false;           {Reset pre-condition}
025 end;
026
027 begin {Main body calls AtTrueABC at intervals of (X2-SmallTime)}
028   Initialize(X1, X2, Y, Z);           {gets timing data}
029   APrime := false;                   {set initial values}
030   A := true;
031   B := false;
032   NextTime := Time + X2 - SmallTime;
033   repeat
034     if AtTrueABC then ReportEventDetected;
035     repeat TimeNow := Time until TimeNow >= NextTime;
036     NextTime := NextTime + X2 - SmallTime;
037   until false;
038 end

```

To fulfill the definition, the choice of the value of constant `SmallTime` in the code is crucial. Even if we could be assured that A, B, and C would be polled at intervals of exactly `x2`, the requirements for intervals `y` and `z` could not be fulfilled. This is because the value of A might have changed from false to true at any time between two successive polls. Thus, we must add delays of at least `y` and `z` when polling B and C, respectively. Yet the values of B and C are not required to remain valid for any longer than `y` and `z`. That would mean the delay timing for `y` and `z` would have to be perfect, which is not achievable with software. Therefore, `SmallTime` is used to increase the polling rate enough to ensure that B and C are polled sufficiently close to the time that A changes from false to true. `SmallTime` must be greater than the maximum amount of error that may occur in the

timing of the polls. Specifically, `SmallTime` must be greater than the maximum delay between the polling of B and A, or A and C, *in excess of y and z*, plus any variability between successive calls of `AtTrueABC`. If a value of `SmallTime` that is much smaller than the value of `x2` cannot be found, then the specifications are not feasible.

In order to validate the above code, we show that:

- the fulfillment of PART 1 of the definition implies that the code will detect the event
- the detection of an event by the code implies the fulfillment of PART 2 of the definition.

INFORMAL PROOF

PART 1 Event Detected Once

The code will detect an event whenever the conjunction AP_{prime} and not $PrevA$ and A and $PrevB$ and C in line 023 evaluates to true. Therefore, we show that the truth of each element of the conjunction in line 023 follows from the definition. Since $AtTrueABC$ (line 011) is called every $x2 \cdot SmallTime$ units of time, there exists exactly one time T : $t6 \leq T < t6 + x2 \cdot SmallTime$ when line 020 is executed.

- A A is polled at time T . Since $t6 \leq T < t6 + x2 \cdot SmallTime < t8$, A is true at time T .
- C C is polled after T , and after the delay of line 021. Let this be time c . The delay will be at least z , and less than $z + SmallTime$. Since $SmallTime < x2$, we have $t7 = t6 + z \leq T + z \leq c < T + z + SmallTime < t6 + x2 \cdot z = t9$.

Therefore, C is true at time c .

AP_{prime} According to the definition, A' is true for an interval of at least $x1$, at some time prior to A becoming true. By the program assumption, $x1 \geq x2$, so there must be a time p when line 019 is executed such that $t1 + SmallTime \leq p \leq t2$, making AP_{prime} true. AP_{prime} would then remain true until A is polled and found to be true, and line 024 is executed. This will, of course, happen shortly after time T , but only after the execution of line 023, which causes the report of the event. Let pA be the time of the last poll of A prior to time T . By the definition of T above, $pA < t6$. Since $SmallTime < y$, $pA > t3$. Therefore, A is false at time pA . Any other polls of A , between times p and pA , happen after time $t1 + SmallTime$ and before $t6$. Therefore, all such polls of A are false, and AP_{prime} will be true when line 023 is evaluated after time T .

not $PrevA$ The reasoning is the same as above for time pA .

$PrevB$ This is the value of B obtained in the previous call to $AtTrueABC$. Let the time of that poll be b . By the specification of $SmallTime$, we are assured that $(T - x2 \cdot y) < b \leq (T - x2 + SmallTime - y)$. Since $t6 \leq T < t6 + x2 \cdot SmallTime$, we have $t3 < b \leq t5$.

Given the above valuations, line 023 will evaluate to true, causing the event to be detected upon the return from the function $AtTrueABC$. Since

AP_{prime} is reset to false prior to that detection, there will be only one detection of the event until the appropriate conditions are fulfilled once again.

Event Detected PART 2

Again, let T be the time that line 020 executes, immediately prior to the detection of the event.

Let:

```

t6 = T;
t7 = t6 + z;
t8 = t6 + x2;
t9 = t6 + x2 + z;
t5 = t6 - y;
t4 = t6 - x2;
t3 = t6 - y - x1;
t2 = t6
t1 = Min[t2 - x1, (the time when APprime was
set to true) - SmallTime]

```

These values satisfy the inequalities on $t1$ - $t9$ from the definition. Now we produce the times referred to in PART 2 based upon the polls of the values that satisfied the conjunction in line 023:

```

∃t[(t1 ≤ t < t2) ∧ A'(t)] and
    [Let t = (the time when APprime was set to
    true by line 019)]
∃t[(t1 ≤ t < t6) ∧ ¬A(t)] and
    [Let t = (the time of the previous execution
    of line 020 prior to t6)]
∃t[(t3 ≤ t < t5) ∧ B(t)] and
    [Let t = (the time of the previous execution
    of line 017 prior to t6)]
∃t[(t6 ≤ t < t8) ∧ A(t)] and
    [Let t = T]
∃t[(t7 < t ≤ t9) ∧ C(t)].
    [Let t = (the time of the execution of line
    022 after t6)]

```

This satisfies both parts of the definition; thus, the program implements the definition.

7. Other Implementation Issues

The example program has implemented the fully extended form of the event descriptor for polled variables. If A were interrupt driven, the code and validation would be more straightforward. The only added complexity would be that, when the value of B changes, the old value and the time of its change must be saved. Then, in evaluating whether the event has occurred, the old value of B must be used if time y has not elapsed since the change.

Since the extended event descriptor is intended to replace the SCR descriptor and definition, we show that it can still express the simpler form of the SCR notation. This is accomplished by noting that $[@T_{x,x}(\neg A \Rightarrow A) \text{ WHEN}_{>0}(B)] \vee [@T_{x,x}(\neg A \Rightarrow A) \text{ WHEN}_{<0}(B)]$, under our definition, reduces to $@T(A) \text{ WHEN}(B)$, with epsilon values of $\pm x$, under the SCR definition.

The above example assumes that all values are polled. To demonstrate the impact of polling versus interrupt schemes, we provide the following examples of code fragments that implement the somewhat simplified expression $@T_x(A) \text{ WHEN}_x(B) \text{ WHEN}_x(C)$. The reader may wish to refer back to Figures 2, 3, and 4 for illustrations of the timing schemes.

- a. "A" polled. The code must alternately poll B, A, and C, waiting for the sequence:

```
B is True.
A is False.
[C is any value.]
[B is any value.]
A is True.
C is True.
```

An alternative approach would allow polling of B until it becomes true, and then beginning the alternate polling of A and B. C then need only be polled after A and B have met the criteria. Particularly where B is interrupt driven, this would mean that no polling at all need be done until B's interrupt code is called.

The following is a sample PASCAL code fragment which implements the first-mentioned approach. External code would be required to execute this fragment at least every x units of time.

```
...
(A is initially True; B is initially
False)
OldA := A;
OldB := B;
PollB(B);
PollA(A);
PollC(C);
if A and OldB and C and not OldA then
PerformAction;
...
```

If optimization to minimize unnecessary polling is desired, the code complexity can be increased to

provide "short circuit" evaluation. It should be noted that, if the language provides short-circuit evaluation of boolean expressions statements, function calls could be used for polling, achieving similar results. However, this would make tracing and verification of polling sequences more complex, and should probably not be used. The following code implements the "short circuit" evaluation.

```
...
(B is initially False)
OldB := B;
PollB(B);
if OldB then
begin
  OldA := A;
  PollA(A);
  if A and not OldA then
  begin
    PollC(C);
    if C then PerformAction;
  end
end
else
begin
  PollB(B);
  if B then PollA(A);
end
...
```

- b. "A" interrupt driven. When A becomes true, the interrupt procedure must check the most recent value of B, and then poll C. If both B and C are true, the event is triggered.

Note: As interrupt procedures are not defined in Standard PASCAL, we shall use the rules of Borland Turbo PASCAL [BORL 91]. The parameter list will be a single type, rather than Borland's enumerated list of CPU registers. An interrupt suspends execution of code anywhere in the program (excepting higher-priority interrupt handlers) and causes execution of the interrupt procedure. Interrupt procedures have access to all global variables.

The following code implements such a procedure. Here, B is a global variable which is polled externally, at least once every x units of time. A and C might be global variables, if their values are required elsewhere.

```

procedure Ainterrupt(Reg :
  RegistersType);
  interr pt; (This procedure is
    triggered any time the monitored value
    of A changes)

var A, C : Boolean;
  procedure InterpretRegisters(Registers
    : RegistersType, A : Boolean);
  begin
    ... {Determines correct value for A}
  end;

begin
  InterpretRegisters(Reg, A);
  if A and B then
  begin
    PollC(C);
    if C then PerformAction
  end
end;

```

In all of the above examples, if *B* and/or *C* are interrupt driven, their interrupt procedures will merely update global variables. As stated in section 4, the interrupt priorities will be: $C > A > B$.

8. Evaluating the Descriptor

The following table adds our descriptor to those covered in Section 4.

The goal of generality is fully met by our proposed extensions of the event descriptor, covering the sequencing and input behavior issues. As demonstrated in the simple coding examples, the basic form of the new descriptor is implementable with simple variable checking in a single conditional statement. The more complex form is also demonstrated without undue code complexity. We also demonstrate the means to implement both polled and interrupt-driven inputs. Finally, we demonstrate a verification based on a code walk-through type proof.

9. Further Work

The work presented here has been produced within the context of a larger work concerning the tracing between Parnas (SCR) style-specifications and code, using the *variable flavor* annotations of [HOWD 90]. During the implementation of "toy" problems, it became apparent that previous definitions of event descriptors were unsatisfactory, as the descriptors did not allow precise specification of certain useful but complex events. Before we could trace, we required a working definition of what code is required to actually implement both simple and complex event descriptors. The production of such a descriptor is the function of this paper. Our next step will be to identify what sort of annotations should be included by the coder to document the implementation.

	Original SCR [NRL 88]	Event Classes [SCHO 90]	Extended Descriptor
Generality:			
Sequencing of WHEN condition-			
Before event	Y	Y	Y
After event	Y	N	Y
Both	N	N	Y
Either	Y	N	Y
Time delays	Y	N	Y
Input Behavior-			
Bouncing	N	N	Y
Oscillating	N	N	Y
Implementation:			
Complete definition of event	N	Y	Y
Simple form	Y	Y	Y
Complex form			Y
Polled / Interrupt variables			Y
Verification:	N	~	Y

```

procedure Ainterrupt(Reg :
  RegistersType);
interrupt; {This procedure is
  triggered any time the monitored value
  of A changes}

var A, C : Boolean;
  procedure InterpretRegisters(Registers
    : RegistersType, A : Boolean);
  begin
    ... {Determines correct value for A}
  end;

begin
  InterpretRegisters(Reg, A);
  if A and B then
    begin
      PollC(C);
      if C then PerformAction
    end
  end;
end;

```

In all of the above examples, if *B* and/or *C* are interrupt driven, their interrupt procedures will merely update global variables. As stated in section 4, the interrupt priorities will be: $C > A > B$.

8. Evaluating the Descriptor

The following table adds our descriptor to those covered in Section 4.

	Original SCR [NRL 88]	Event Classes [SCHO 90]	Extended Descriptor
Generality:			
Sequencing of WHEN condition-			
Before event	Y	Y	Y
After event	Y	N	Y
Both	N	N	Y
Either	Y	N	Y
Time delays	Y	N	Y
Input Behavior-			
Bouncing	N	N	Y
Oscillating	N	N	Y
Implementation:			
Complete definition of event	N	Y	Y
Simple form	Y	Y	Y
Complex form			Y
Polled / Interrupt variables			Y
Verification:	N	~	Y

The goal of generality is fully met by our proposed extensions of the event descriptor, covering the sequencing and input behavior issues. As demonstrated in the simple coding examples, the basic form of the new descriptor is implementable with simple variable checking in a single conditional statement. The more complex form is also demonstrated without undue code complexity. We also demonstrate the means to implement both polled and interrupt-driven inputs. Finally, we demonstrate a verification based on a code walk-through type proof.

9. Further Work

The work presented here has been produced within the context of a larger work concerning the tracing between Parnas (SCR) style-specifications and code, using the *variable flavor* annotations of [HOWD 90]. During the implementation of "toy" problems, it became apparent that previous definitions of event descriptors were unsatisfactory, as the descriptors did not allow precise specification of certain useful but complex events. Before we could trace, we required a working definition of what code is required to actually implement both simple and complex event descriptors. The production of such a descriptor is the function of this paper. Our next step will be to identify what sort of annotations should be included by the coder to document the implementation.

10. Acknowledgment

We wish to express our appreciation for the reviews, suggestions, and advice given by Bruce Labaw and Anne Rose of the Navy Research Laboratory, and Jo Atlee and John Gannon of the University of Maryland.

11. Appendix: Extended Example Code

The example implementation of event-detecting code show in Section 6 requires only minor extensions to allow for values of x_1 that are greater than x_2 . This situation would arise where the truth of a precondition (A') needed to be assured over a length of time greater than the

polling interval x_2 . The following code uses a variable $APDuration$, which represents an upper bound on the amount of time that A' has been true. Although the explanation is too complex for inclusion here, the comparison of $APDuration$ and $X_1 - X_2$ in line 031 ensures that the code will recognize that the precondition is satisfied whenever A' is true for an interval of at least x_1 , and that it will only detect the event if A' is true at least once in every sub-interval of length x_2 during an interval of length x_1 . Thus, this code fulfills the definition of $@_{T_{x_1, x_2}}(A' \Rightarrow A)$ WHEN $\neg_y(B)$ WHEN $\neg_z(C)$.

```
001 program RealTimeProg;
002 var
003   A, APrime, B, C : Boolean;
004   TimeNow, NextTime, PollTime, APDuration, X1, X2, Y, Z : TimeType;
005 const
006   SmallTime : TimeType = {SmallValue};    {See explanation below}
007
008 {functions Time, and procedures Initialize, Delay, PollA, PollB,
009 PollC, PollAPrime, and ReportEventDetected not shown}
010
011 function AtTrueABC : Boolean;              {Returns true if event occurs}
012 var
013   PrevA, PrevB, PrevAPrime : Boolean;
014 begin
015   PrevA := A;                             {Save previous values of A & B}
016   PrevB := B;
017   PollB(B);                               {Poll A, B, C with delays of Y and Z}
018   Delay(Y);
019   if APDuration <= X1 - X2 then            {Check precondition only
020   begin                                    if not already satisfied}
021     PrevAPrime := APrime;
022     PrevTime := PollTime;
023     PollAPrime(APrime);
024     PollTime := Time;
025     if APrime then
026       APDuration := APDuration + PollTime - PrevTime;
027   end;
028   PollA(A);
029   Delay(Z);
030   PollC(C);
031   AtTrueABC := (APDuration > X1 - X2) and
032     not PrevA and A and PrevB and C;
033   if A or not APrime then APDuration := 0; {Reset pre-condition}
034 end;
035
036 begin {Main body calls AtTrueABC at intervals of (X2-SmallTime)}
037   Initialize(X1, X2, Y, Z);               {gets timing data}
038   APDuration := 0;                        {set initial values}
039   APrime := false;
040   PollTime := Time;
```

```

041  A := true;
042  B := false;
043  NextTime := Time + X2 - SmallTime; (Determine polling interval)
044  repeat                               (main loop--calls polling routine)
045      if AtTrueABC then ReportEventDetected;
046      repeat TimeNow := Time until TimeNow >= NextTime;
047      NextTime := NextTime + X2 - SmallTime;
048  until false;
049  end

```

12. Literature

[BORL 91] *Turbo Pascal for Windows Programmer's Guide*, Borland International, Inc., 1991.

[FAUL 89] *State Determination in Hard-Embedded Systems*, Stuart Roland Faulk, doctoral dissertation at UNC, Chapel Hill, June 1989.

[HENI 80] "Specifying Software Requirements for Complex Systems: New Techniques and Their Application", Kathryn L. Heninger, *IEEE Transactions on Software Engineering*, Vol. 6, No. 1, January 1980, pp. 1-13.

[HOWD 90] "Comments Analysis and Programming Errors", William E. Howden, *IEEE Transactions on Software Engineering*, Vol. 16, No. 1, January 1990, pp. 72-81.

[NRL 88] *Software Requirements for the A-7E Aircraft*, Alspaugh et. al., Naval Research Laboratory. First released in November 1978, second and final release December 1988.

[SCHO 90] *The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems*, A. John van Schouwen, Technical Report 90-276, Queen's University, May 1990.

Economical Development of Complex Computer Systems

Thomas C. Choinski, M.B.A.

Senior Project Engineer

Naval Undersea Warfare Center Detachment, New London

(203) 440-5391

"Knowing where you've come from and where you are is essential to knowing how to get where you want to go. Developing a new generation of products is a lot like taking a journey into the wilderness. Who would dream of setting off without a map?"¹

Steven C. Wheelwright and W. Earl Sasser, Jr.

INTRODUCTION

This paper proposes a product development methodology (PDM) for complex systems evolving within the current economic climate of the United States, as well as the unstable state of world affairs envisioned throughout the decade. The PDM facilitates the development of systems that are "multipurpose, flexible, highly mobile, and incorporate maximum bang for the buck."²

Ironically, the unstable nature of the development environment within the defense community parallels the one encountered by the commercial sector over the last 20 years. Successful companies have responded by adopting a product development methodology that adapts to ever changing market demands and the concern for near term returns (profit).

The PDM exploits lessons learned from the commercial market analogy to establish a flexible, low risk, cost effective approach for technological progress. The approach suits systems development, especially those involving complex mission critical computer systems.

Examination of the commercial product development process reveals a strategy that can achieve the procurement flexibility needed by DoD. This strategy concentrates on leveraging the state-of-the-art in a cost effective manner. The strategy also addresses risk management.

The key to the technological success of this strategy relies on an incremental development process. The IBM PC serves as a perfect example. The i486 based PC resulted from successful sales of the 286, 386SX and i386 based versions. Incremental upgrades enabled IBM to respond to changes in market demand as well as facilitate the transition of the state-of-the-art. In this manner, IBM attained strategic flexibility.

The discussion starts at a basic level and progresses to a macroscopic perspective. This paper contains four parts: adaptation of a commercial approach, incorporation into an overall risk management scheme, application to an open architecture transition, and a summary.

The paper recommends using open architectures and commercial off-the-shelf (COTS) items to implement the incremental improvements outlined by the PDM. The summary includes guidelines for successful product development, as well as ideas for future work in this area.

ADAPTATION OF A COMMERCIAL APPROACH

Decreasing commercial product life cycles have required technologies to be developed at faster rates.³ As a result, companies have devoted more effort to the product development process. The process differs from company to company; however, the *high tech* arena focuses on the time to market. Shortening the time to market enables a company to increase market share, adapt product characteristics to market needs, enjoy high margins typically encountered in the beginning of a product's life cycle, and shorten the payback period.⁴

This focal point requires a company to make decisions in what Preston Smith refers to as the "fuzzy front end" of the product development process.⁵ Lack of quantitative information and organizational structure characterize the ambiguity of the "fuzzy front end." Decisions made at this time greatly affect the product's evolution. The expensive nature of changes made down the road heightens the significance of these up front decisions. As a product progresses from planning, through design, production, test and delivery the cost to correct an error increases.⁶

Smith offers a simple decision analysis technique to attack problems in the "fuzzy front end." The technique concentrates on the interrelationships between time, development cost, performance and profit. He prefers an approach based on estimates generated quickly. Smith believes complicated estimation tools waste time and lead to a false impression of the accuracy of the available data. His book presents several examples to demonstrate the merit of his approach. Therefore, this paper concentrates on the adaptation of Smith's ideas to complex systems.

At first glance, the lack of the profit motive within the government appears to create an obstacle to the application of Smith's approach to the development of complex systems. However, consider the savings the government can pursue when building systems. This viewpoint establishes the profit motive; when systems cost less, profits from savings follow. In other words, life cycle cost savings create a profit. The analogy between profit and cost savings permits a modification to Smith's model for product development in the defense sector. Figure 1 illustrates the product development framework that results when life cycle cost replaces profit in Smith's model. The arrows indicate relationships between the areas identified in the circles.

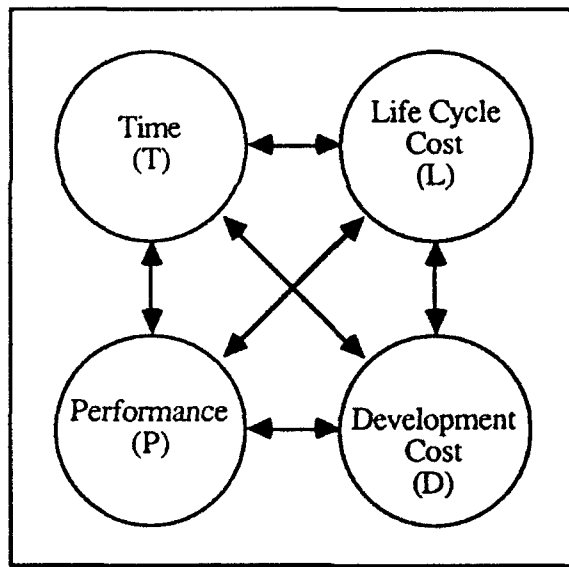


Figure 1. Product Development Framework

Note, this model separates development costs from life cycle costs. This definition differs from the definition of life cycle cost used in Naval acquisitions. For Naval acquisitions, life cycle cost is the sum total of the direct, indirect, recurring, non-recurring, and other related costs incurred, or estimated to be incurred in the design, research and development (R&D), investment, operation, maintenance, and support of a product over its life cycle.⁷

The product development framework fosters decisions based on time, performance, development cost and life cycle cost tradeoffs. To achieve savings, the product development framework must assume a baseline for time, performance and cost. An existing system functions as the baseline. The baseline system establishes cost and performance ceilings. Measure time, performance and cost in terms of the incremental contribution to the baseline system when developing new products for existing systems.

This paper uses qualitative reasoning to demonstrate the utility of the product development framework. A color coding scheme depicts the incremental contribution for each area. Ideally, the color coding scheme would use a traffic light pattern. Picture red for an undesirable rating, yellow for an indeterminate condition and green for a favorable estimate. Figure 2 exhibits the alternate color coding scheme used in this paper to facilitate duplication of the material.

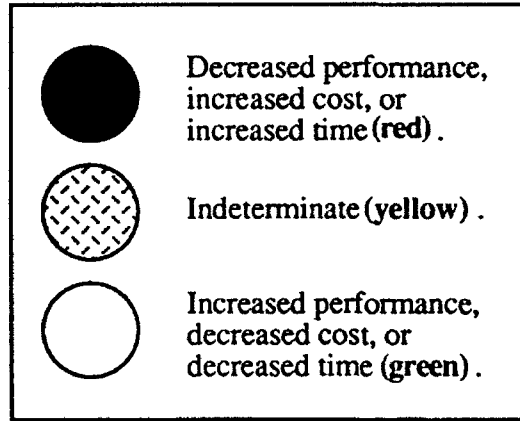


Figure 2. Color Coding Scheme

A quantitative analysis would eventually replace qualitative reasoning. The progression of time enables the analysis to improve as more information becomes available and decisions are revisited. Hence, the quantitative analysis gets fine tuned as the product becomes well defined.

This product development framework facilitates the assessment of life cycle costs, development costs, and development time for specific performance requirements. Continued appraisal will result in a set of performance requirements that meets cost goals.

One problem not represented directly in the framework is the difficulty mustering support for the acquisition of systems on the basis of life cycle cost. Opponents can attack the fidelity of forecasts beyond a 5 year period.

However, a shorter product development cycle addresses this problem by trimming the payback period. The payback period is the time it takes to recoup the initial development cost through life cycle cost savings. Reduced payback periods strengthen life cycle cost estimates. The incremental product development approach capitalizes on condensed payback periods.

The framework addresses issues on a discrete product basis. Examples of discrete products include: disk drives, power supplies, and stand alone computers. In contrast, complex computer systems represent an amalgam of discrete computer products. They require a technique that weighs each decision on a macroscopic level. The four element diagram cannot guide complex decisions without a higher level of abstraction. The next section outlines the higher level.

RISK MANAGEMENT FOR COMPLEX COMPUTER SYSTEMS

Many discrete technical approaches compete for attention in the "fuzzy front end" of complex computer systems development. The aforementioned product development framework expedites decisions on a case by case basis, but cannot manage a complex computer system in its entirety. A useful methodology must provide a map for macroscopic considerations. This consideration differentiates Smith's commercial technique from complex systems development. Nevertheless, Smith's framework forms a foundation for the map used in the higher level of abstraction.

Basically, the methodology sets the stage for each discrete approach to compete in terms of time, cost and performance. A three step process establishes a clear path from the "fuzzy front

end" through product definition to a low risk development scenario. Figure 3 illustrates this process for the development of a complex computer system on the basis of cost.

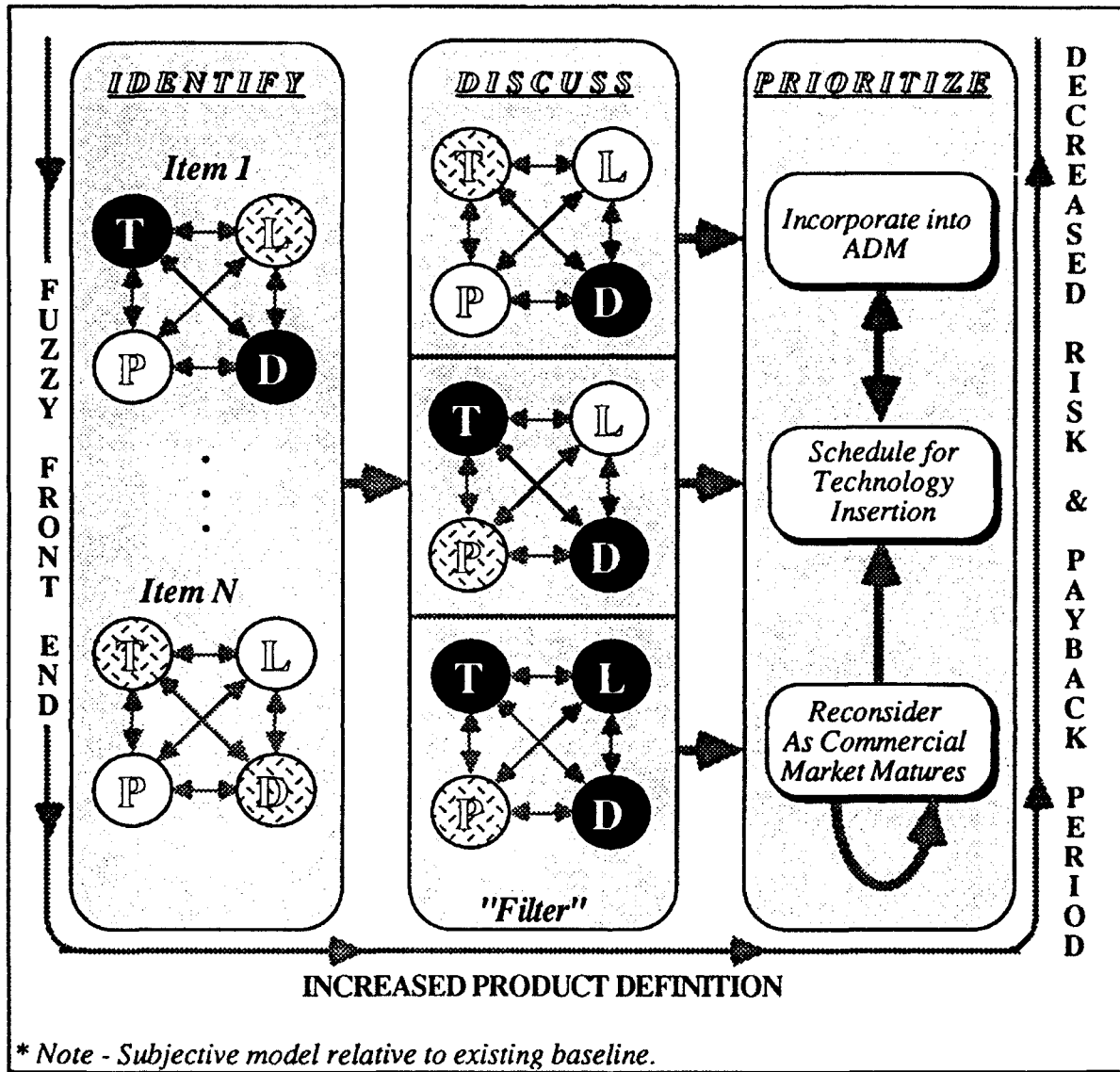


Figure 3. Product Development Methodology

The first step identifies each discrete candidate. Identification requires at least a subjective description in terms of time, performance, life cycle cost and development cost. In fact, subjective approaches accelerate the process. A plethora of candidate approaches typically overwhelms the front end of complex computer system development. A detailed quantitative analysis of each would consume time and money.

A RAND study of process plants demonstrates the lack of accuracy of data in the "fuzzy front end." Process plant estimates generated on the basis of R&D data alone, can easily overrun budgets by 100%. As the level of project definition and quantity of engineering data increase, overruns decline to about 10% at a full cost design stage.⁸

Ensure early efforts focus on the rapid development of high potential products, rather than up front detailed cost analyses. Get products into existing systems quickly. Keep the up front

analysis simple to reduce development costs. Mitigating costs reduces risk. In the long term, the incremental product development methodology promotes a diversified development portfolio.

Breaking down the complex system development into a step-by-step sequence of limited challenges contains risk and cost. Northern Telecom's venture from analog into digital switching systems for the telecommunications industry serves as an example. Instead of going for the local DMS 100 switch right away, the company started with the development of a PBX (private branch exchange), which gave it a base for understanding important new technologies, digitization techniques, advanced programming languages, and network design. Treating the effort as a step-by-step sequence of more limited challenges allowed Northern Telecom to contain its development risk and keep development costs from going through the roof.⁹

The second step involves a discussion of each candidate approach. The discussion includes establishing fundamental criteria for advanced development, technology insertion and future consideration. Discussion challenges the subjective nature of the data.

The third and final step sorts the candidates into those considered for immediate advanced development, future technology insertion, and further consideration. At this part of the development process the high priority candidates require a detailed quantified analysis. Depending on cost goals, products can move to and from the advanced development model (ADM) and technology insertion. Existing ADM products replaced by candidates from the technology insertion area act as contingencies; they create a backup in case of product failures.

The identification, discussion and prioritization (IDP) of discrete candidates lead to system definition. Figure 3 shows how development proceeds from the "fuzzy front end" to clear-cut product definition. The process mitigates risk by giving priority to approaches that yield the biggest cost savings with the shortest payback period. As time progresses the product becomes well defined and information is available to make decisions quantitatively rather than qualitatively.

APPLICATION OF THE PDM TO AN OPEN ARCHITECTURE TRANSITION

Traditionally, combat system development requires a quantum leap in performance. Figure 4 exemplifies the computing throughput required by expanding sensor array configurations.

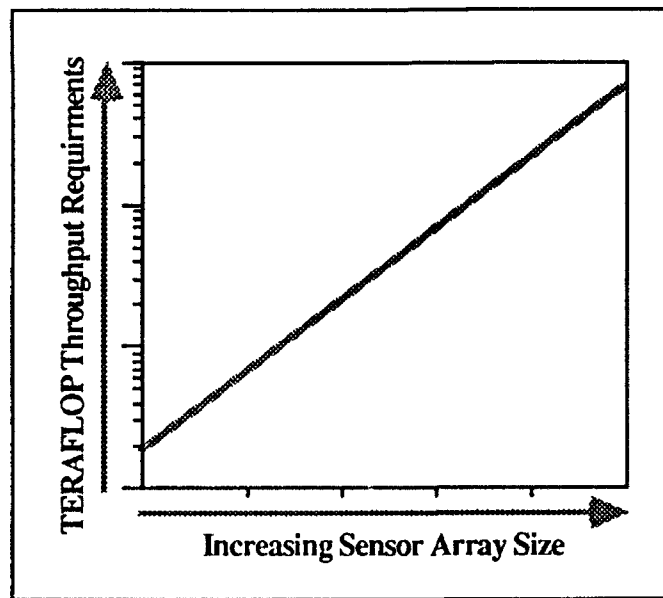


Figure 4. Quantum Leap in Processing Requirements

In addition to greater computational requirements, this trend also creates demands for faster communication links and denser memory configurations. One approach to meeting these demands involves the incorporation of an open architecture. Open architectures leverage fast paced commercial technology development by maintaining compatibility with commercial standards.

Case histories show building off existing foundations of core technologies generates success in the commercial sector. Companies that focus new products on extensions to a single key technology are far more successful than those that pursue technical diversity. "The best opportunities for rapid growth come from building an internal critical mass of engineering talent in a focused technological area, yielding a distinctive core technology that might evolve over time, to provide a foundation for the company's product development."¹⁰

Accordingly, the transition from an existing sensor processing system to an open architecture based system offers a prime example of the utility of the PDM proposed in this paper. The combination of the PDM and open architecture philosophies facilitates future technology upgrades for the sensor system. Hardware and software commonality contain costs.

Figure 5 shows an open architecture for a sensor processing system. The key features of this architecture are the sensor distribution network, the data distribution network and the common processing cabinets. The common processing cabinets fit into the open architecture scheme by utilizing a commercially available bus architecture for the backplane. Any vendor can integrate equipment into the system as long as they adhere to the interface standards.

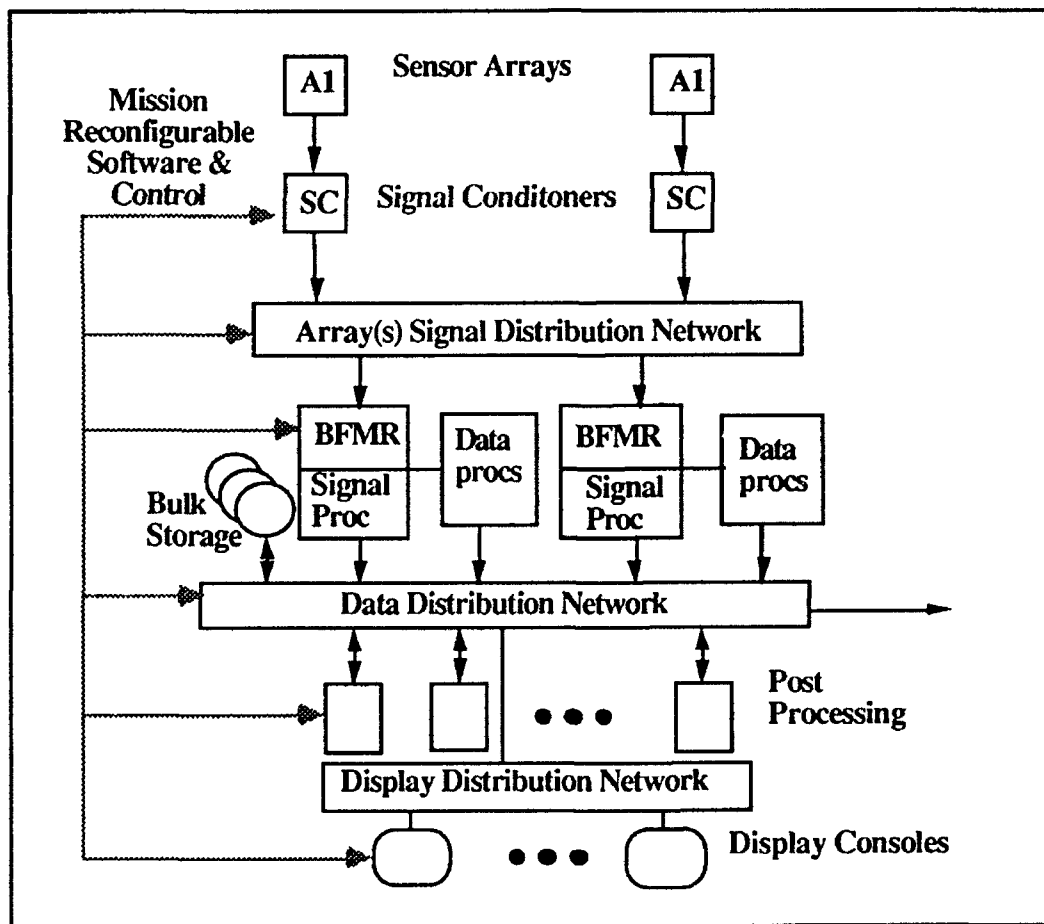


Figure 5. Notional Open Architecture

The lack of mature standards poses an obstacle to the implementation of open architectures. For example, many of the Next Generation Computing Resources (NGCR) initiative's interface standards have not been written. Therefore, cost and performance are indeterminate.

In addition, many existing combat systems do not possess open architecture attributes. On one hand, existing system baselines minimize development costs. On the other hand, open architectures facilitate life cycle development. The current fiscal environment within DoD does not favor system development programs with a high cost profile. Nonrecurring engineering funds are shrinking. An incremental transition from an existing system to open architecture would spread out the cost and mitigate the risk. The incremental approach also advances the long term goals of open architectures.

The product development methodology proposed in this paper helps attain this goal. Using an existing system as a baseline, the transition takes a low risk path to incrementally integrate open architecture concepts into the complex computer system. Figure 6 depicts this concept for a generic sensor processing chain.

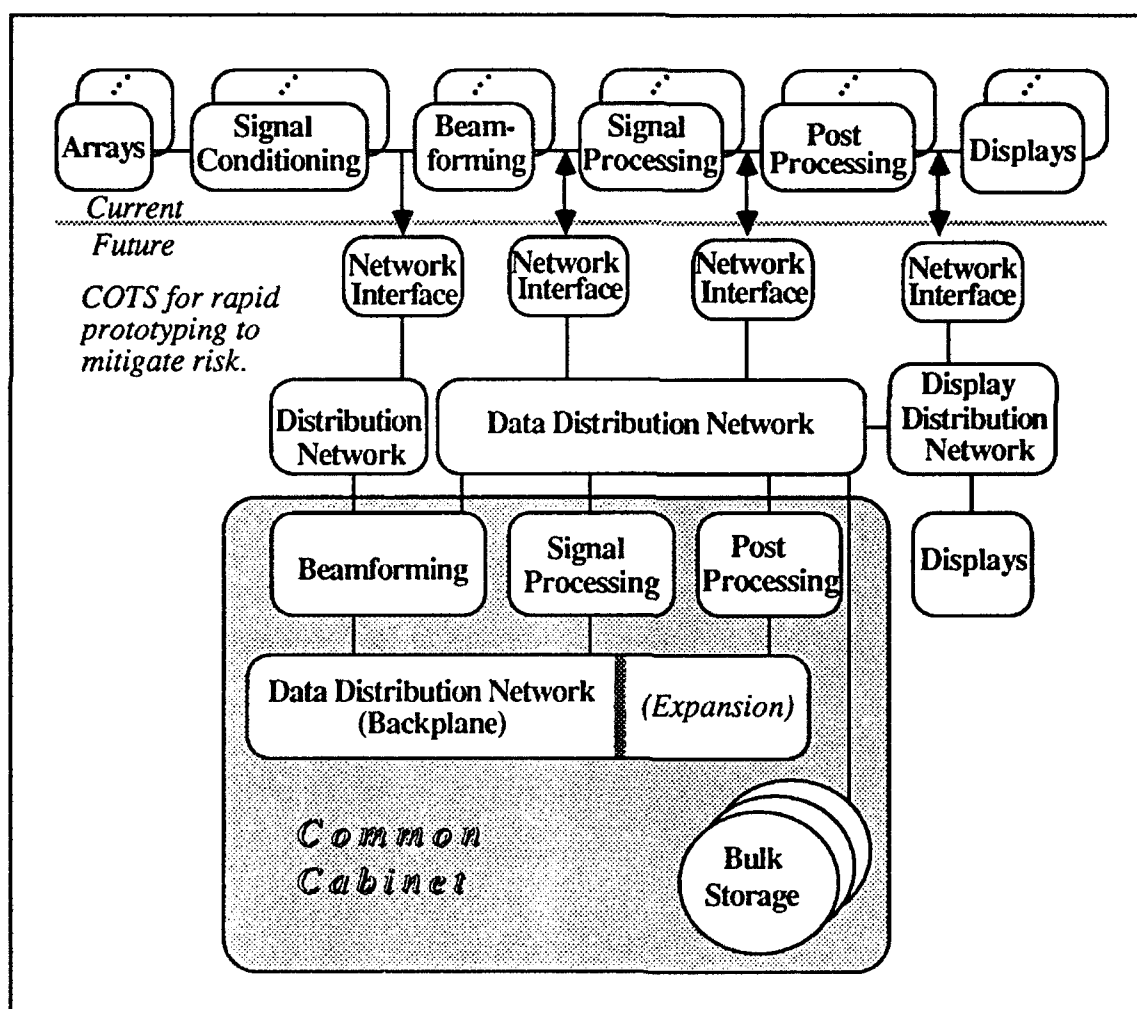


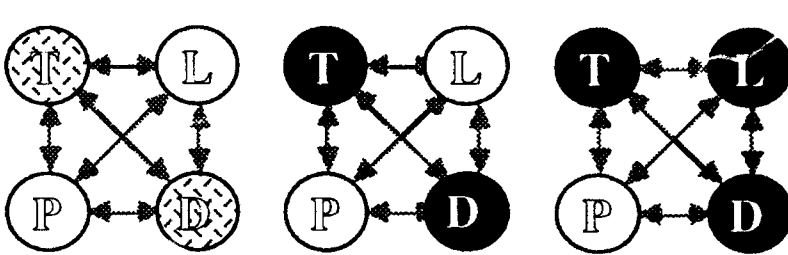
Figure 6. Incremental Approach for Open Architectures

The network interface builds an open architecture upon the strong foundation of the baseline sensor processing chain. First, tap into the processing chain. Next, insert the subsets of the open architecture through the network interface units. For example, evaluate a novel beamformer

by bypassing the existing beamformer. Gradually change the system as technology becomes available at a suitable cost. Eventually, the open architecture replaces the baseline sensor chain.

The PDM facilitates consideration of various implementations for each functional block. Commercial off-the-shelf equipment serves as an excellent implementation for initial product development. COTS equipment maintains the cost for initial test and evaluation. Test the COTS prototypes for shock, vibration, temperature, etc., and deploy the equipment with acceptable performance. Militarize the COTS equipment if environmental test results fall short of expectations. Risk mitigation occurs because the government expends additional capital only for verified performance. In addition, the availability of the baseline system serves as a contingency to reduce the risk of product failure.

Table I clarifies the utility of the PDM for a next generation sensor system. In this scenario, a sensor system already in production serves as the baseline. Generic candidates for technology insertion includes today's technology, near term upgrades, and projected future commercial technology.



Processing Type	Today's Technology	Technology Near Term (<1 Year)	Future (>5 Years)
Conventional Beamformers	4,000 MIPS	4,000 MIPS	40,000 MIPS
Adaptive Beamformers	NA	0.3 GFLOPS	5 GFLOPS
Signal Processing	0.3 GFLOPS	2 GFLOPS	21 GFLOPS
Data Processing	70 MIPS	70 MIPS	300 MIPS
Data Processing	35 68030s	35 68030s	130 68040s
Data Processing & I/O	130 68030s	130 68030s	500 68040s

* All units represent effective capability on a per cabinet basis.

Table I. PDM Application to Open Architecture Technology Insertion

Note the equipment undergoing advanced development functions as an excellent augmentation to the baseline system. The technology insertion category would include near term signal processing and adaptive beamforming technologies. Future oriented technologies, which include parallel processors (e.g., iWarp, Paragon, Connection Machine) repackaged in multichip modules, do not demonstrate a definitive payoff. The PDM suggests reconsideration of these technologies when the commercial market brings down the cost. In the interim, develop the network interface units to facilitate the insertion of the advanced technologies as the market matures.

SUMMARY

This paper proposes an economical product development methodology for complex computer systems. The PDM exploits the commercial market analogy to establish a flexible, low risk, cost effective approach for technological progress. The strategy pursues the state-of-the art while addressing risk management. The key to the methodology is an incremental development process.

The PDM assesses discrete candidates and simplifies macroscopic decisions. The process provides the opportunity to pursue the state-of-the-art while concentrating on choices that emphasize low risk, cost savings, and short payback periods.

Several guidelines enhance the chances of attaining this objective:

1. increase performance/technology in an incremental fashion,
2. use subjective decision techniques to eliminate poor candidates from the start,
3. fine tune detailed quantitative analyses as the product becomes well defined,
4. create a diversified development portfolio directed at a quantum leap in technology,
5. use COTS equipment for rapid prototyping,
6. build products quickly to reduce development costs, and
7. cultivate products with short payback periods.

The methodology has particular application to complex mission critical computer systems. An open architecture transition illustrates the utility of the IDP product development methodology. The PDM sets priorities for a sensor processing chain by subjectively identifying the time, cost and performance characteristics of candidate technologies.

System engineers can use the product development methodology described in this paper by:

1. creating a detailed handbook to guide decisions,
2. devising an expert system to expedite the selection for technology insertion,
3. applying software tools which refine the hierarchical decision process, and
4. using the PDM as a basis for developing complex mission critical computer systems.

The PDM integrates the choices input at lower levels into higher level systems engineering decisions. System engineers could clearly define the affect of the paths from subsystem to system level design. Each candidate at the lower level contributes to overall cost, performance, schedule and risk assessments. In this manner, the PDM enables an efficient approach for systems engineering.

ACKNOWLEDGEMENT

The author would like to acknowledge the notional architecture concepts conceived by Dr. José Muñoz and the transitional approach to open architectures originated by Steve Harrison. In addition, the author would like to thank all the people who reviewed this paper, especially James Hall Jr. and Roger Read.

REFERENCES

- 1 Steven C. Wheelwright and W. Earl Sasser, Jr., "The New Product Development Map," Harvard Business Review, Number 3, May-June 1989, p. 113.
- 2 Commander Bruce Lemkin, U.S. Navy, "The New Leader of the Pack," Proceedings of the U.S. Naval Institute, Volume 117/6/1,060, June 1991, p. 44.
- 3 Christoph-Friedrich von Braun, "The Acceleration Trap," Engineering Management Review, Volume 19, Number 3, Fall 1991, pp. 13-19.
- 4 Milton D. Rosenau, "Speeding Your Product to Market," Engineering Management Review, Volume 17, Number 3, September 1989, pp. 27-40.
- 5 Preston Smith, Developing Products in Half the Time (New York: Van Nostrand Reinhold, 1991).
- 6 Joseph T. Vesey, "The new Competitors: They Think in Terms of Speed to Market," Engineering Management Review, Volume 19, Number 4, Winter 1991, pp. 12-17.
- 7 Department of Defense, Life Cycle Cost in Navy Acquisitions, MIL-HDBK-259 (Navy), 1 April 1983, p. 5.
- 8 David Davis, "New Projects: Beware of False Economies," Harvard Business Review, Number 2, March-April 1985, pp. 95-96.
- 9 Edward G. Krubasik, "Customize Your Product Development," Harvard Business Review, Number 6, November-December 1988, p. 49.
- 10 Edward B. Roberts and Marc H. Meyer, "Product Strategy and Corporate Success," Engineering Management Review, Volume 19, Number 1, Spring 1991, p. 6.

Improving Safety Margins in Rate Monotone Scheduling

Radhika Menon

Arkady Kanevsky

Swaminathan Natarajan

Myung Jun Kim

Department of Computer Science

Texas A&M University

College Station, TX 77843

June 19, 1992

Abstract

In this paper we propose several strategies for improving the safety margin of a real-time system using the rate monotone algorithm, by utilizing application characteristics. The rate monotonic scheduling algorithm assumes that all tasks are initiated simultaneously. In this work, we relax this worst-case assumption and determine the optimal initiation times for a 2-task system to increase the utilization bound of the system. It turns out that the achievable utilization depends also on the relationship of task periods. We then investigate this relationship and show how task periods may be modified to further optimize the utilization bound. This results in an increased safety margin of the system. We derive analytically expressions for optimal initiation times and utilization bound for a 2-task system. Extending a similar analytical study to a system with an arbitrary number of tasks is extremely complex. So we develop algorithms using the same ideas and simulation results show a similar increase in the safety margin of the system.

1 Introduction

The rate monotonic scheduling algorithm was introduced by Liu and Layland [3] and is known to be optimal among static, priority driven, preemptive scheduling algorithms for real-time environments, subject to its underlying assumptions. It is optimal in the sense that no other fixed priority assignment rule can schedule a task set which cannot be scheduled by the rate monotone priority assignment. The assumptions include strict periodicity, task independence, constant running times. Some of these assumptions restrict applicability to specific system models. Sha and others [1, 2, 5, 7, 6] have enhanced this algorithm by devising techniques to deal with non-independent task sets, aperiodic tasks, stochastic execution times and resource sharing. These were attempts at making the rate monotone algorithm applicable to different system models, by relaxing some of the assumptions. In our work, the system model is the same, but we use the semantic information about the application to avoid worst case situations. The two characteristics we exploit here are the *task initiation times* and the *task periods*.

Liu and Leyland [3] showed that simultaneous initiation of tasks creates worst-case situations. Subsequent research has continued with this assumption. Similarly current research also assumes that tasks have some fixed user-specified periods. The motivation for this research stems from the fact that in typical practical applications, system designers do have some choices in the selection of the exact periods and initiation times of periodic tasks.

For example, an application such as the space station may contain several kinds of periodic tasks: data acquisition tasks, such as obtaining and recording the values from sensors; situation monitoring tasks, such as the cabin temperature monitoring performed by the life support system; and control applications, such as navigation. The periodicity requirement on data acquisition and situation monitoring is typically of the form "perform at least every 5 seconds". If the system designers actually choose a shorter period because that fits the scheduling, the system performance will only be improved. Also, since the requirement is only that each task be performed periodically, unrelated tasks may not have any restriction on their relative phasing. The time at which the cabin pressure is monitored can be quite independent of the time at which navigation commands are issued. Thus, except in cases where a particular phasing is specified because of dependencies, designers can phase tasks in any way which suits the scheduling. Since most other real-time applications also commonly perform data acquisition, control and monitoring functions, similar flexibility is likely to be available in a wide range of systems.

We utilize this flexibility to obtain two kinds of benefits in the design of the scheduling. The first is to enhance the schedulability of the system by increasing the *utilization bound*. In doing this, it is possible that some task sets which were previously unschedulable may now become schedulable. The second goal of our work is to increase the *safety margin* of the system. If we have determined bounds on the computation times of the tasks in the system, and the analysis shows that the set of tasks is schedulable, it is still desirable to provide for exceptional situations where the computation times of some tasks happen to exceed the computed bound. We use the term safety margin to denote the extent by which the system utilization may increase before the scheduling breaks down and deadlines are missed. By adjusting task initiation times and selecting task periods appropriately, we increase the safety margin of the system, at no run-time cost.

Our work is divided into two parts. The first is an analytical study of a 2-task system. We determine optimal initiation times, methods to modify the task periods and study the effects of these modifications on the safety margin of a 2-task system. However, it turns out that the analytical techniques used do not extend conveniently to multi-task systems. Hence, in the second part of our research, we develop algorithms to determine initiation times for the tasks and to modify the periods in order to improve the safety margin. We evaluate the performance of our algorithms by a simulation study, which involves determining the *breakdown utilization* of the task set as described by Lehoczky, Lam and Ding [1].

The remainder of the paper is organized as follows. In section 2 we introduce the previous enhancements to the rate monotone algorithm to which this paper is a contribution. In section 3 we present the analytical study of a two task system and illustrate by examples how the results can be used. In section 4 we develop algorithms to determine initiation times and modify task periods, in a system with an arbitrary number of tasks. We also describe a simulation study to evaluate the performance of these algorithms and present the results. In section 5 we summarize our results and conclude our work.

2 Background

Liu and Layland [3] developed analytical results on the behavior of the rate monotone scheduling algorithm. In order to do so they made a number of assumptions that include strict periodicity, constant running times, independent task sets and simultaneous initiation of tasks.

The rate monotone algorithm assigns higher priorities to tasks with higher request rates. Such a priority assignment is *optimum* in the sense that no other fixed priority assignment rule can schedule a task set which cannot be scheduled by the rate monotone priority assignment. The request rate of a

task is defined to be reciprocal of its request period. The utilization factor of an n task system is defined to be

$$U = \sum_{i=1}^n \frac{C_i}{T_i}$$

where C_i is the computation time, and T_i is the period of task τ_i , $1 \leq i \leq n$.

They proved that this algorithm can schedule any set of n periodic tasks with processor utilization no larger than $n(2^{\frac{1}{n}} - 1)$ or any task set of any size with processor utilization below $\ln 2 \approx 0.693$ [3]. They show that a *critical instant* occurs when requests of all tasks arrive simultaneously, and this is the worst-case situation i.e. if all deadlines are met during the critical region, the system will always meet all its deadlines.

The Liu and Layland bound decreases monotonically from .83 when $n = 2$ to $\log_e 2 = .693$ as $n \rightarrow \infty$. However, the rate monotone algorithm can often successfully schedule task sets having a total utilization higher than .693. In fact, task sets with utilization as high as .9 are often schedulable. This suggests that the average case behavior is substantially better than the worst case behavior. Lehoczky, Sha and Ding [1] describe an exact schedulability criterion to determine task set schedulability. They perform a stochastic analysis and determine that for uniformly distributed tasks, a breakdown utilization of .88 is a reasonable characterization of breakdown utilization level.

Sha, Lehoczky and Rajkumar [5] address the problem of stochastic execution times. In many applications the worst case execution time is much larger than the average case execution time and a low processor utilization would result if it is to be ensured that the system never becomes overloaded. Hence in order to achieve a reasonable average processor utilization a scheduling algorithm must be able to take care of transient overloads. A period transformation method is developed to enhance the stability of the algorithm.

A real-time system has both periodic and aperiodic jobs. Lehoczky, Sha and Strosnider [2] developed the Deferrable Server (DS) and the Priority Exchange (PE) algorithm, based on the rate monotone scheduling algorithm. Both algorithms give a greatly improved average response time for soft deadline aperiodic tasks while still guaranteeing the deadlines of periodic jobs. The Extended Priority Exchange (EPE) algorithm, an extension of PE was developed by Sprunt, Lehoczky and Sha [7]. Sprunt, Sha and Lehoczky [8] devised the sporadic server algorithm, an improvement over the above algorithms.

Sha, Rajkumar and Lehoczky [6] have developed a priority inheritance protocol and derived a set of sufficient conditions under which a set of periodic tasks that share resources using this protocol can be scheduled using the rate monotone algorithm.

All of the above enhancements are attempts at making the rate monotone algorithm applicable to different system models. Our work however focuses on the avoidance of worst case situations. The key concept in our work is avoidance of critical instants. We first show how this can be done by choosing initiation times of the tasks. Then we show how task periods may be selected to avoid simultaneous task arrivals. We do this for a system of two periodic tasks by analyzing the structure of task arrivals in each time period. We then analyze how these modifications affect the safety margin of the system. Extending the analytical techniques to a general n task system is extremely complex. We describe algorithms to determine initiation times and task periods based on these ideas and determine by simulation the improvement to the safety margin of the system.

3 Analysis of a 2-Task System

3.1 Structure of Task Arrival Patterns

In this section we derive the request patterns over different periods for the case of two periodic tasks. Assume without loss of generality that $T_1 < T_2$. The rate monotonic scheduling algorithm assigns higher priority to the task τ_1 . Let the i th period of a task be the interval between the arrival of the i th request and the deadline for that request. Let I_i denote the i th period of the task τ_2 . Since the rate monotonic algorithm is preemptive and τ_1 has a higher priority, the *available time* for the execution of τ_2 in I_i is the amount of time that remains in I_i after executions of the task τ_1 . Let C_2 denote the available time for the execution of τ_2 in I_i .

If for some i , the execution time C_2 of the task τ_2 is larger than $C_{2,i}$, then the system consisting of these two tasks is not schedulable [3]. The key idea here is to move the starting time of the second task τ_2 so as to maximize the smallest $C_{2,i}$ over all i .

We derive the structure of I_i 's based upon the periodicity of the tasks and the assumption that periods T_1 and T_2 as well as the run-time C_1 are constants. In this section we determine how this structure changes depending on the relative starting points of the two tasks.

The structure of I_i can be characterized by the first request of the first task τ_1 in I_i . Let D_i denote the time interval between the i th request of the task τ_2 and the first request of the task τ_1 in I_i . Let k be the *least common multiple* of T_1 and T_2 divided by T_2 . Hence, $D_k = D_0$.

Observation 1 D_i 's are distinct for all $0 \leq i \leq k-1$.

We can characterize I_i by its D_i . Let D'_i be a sorted list of D_i 's for $0 \leq i \leq k-1$. Let I'_i be a list of I_i 's with I'_i corresponding to D'_i . If the task τ_2 starts δ after the start of the first task τ_1 then for all D_i that are larger than δ , D_i will decrease by δ and for all D_i that are smaller than δ , D_i will increase by $T_1 - \delta$. Let \bar{D}_i be that updated D_i :

$$\bar{D}_i = \begin{cases} D_i - \delta & \text{if } 0 \leq \delta \leq D_i \\ D_i - \delta + T_1 & \text{if } D(i) < \delta \leq T_1 \\ \delta - D(i) & \text{if } -D_i \leq \delta \leq 0 \\ T_1 - D_i + \delta & \text{if } -T_1 \leq \delta \leq -D_i \end{cases} \quad (1)$$

Let \bar{D}'_i 's be a sorted list of \bar{D}_i 's for $0 \leq i \leq k-1$.

Observation 2 If there are i and j such that $D_i = \bar{D}_j$ then $\bar{D}'_r = D'_r$ for all r .

Lemma 1 For all $0 \leq i < k-1$ $D'_{i+1} - D'_i = \frac{T_1}{k}$.

There are two consequences to this lemma. For any starting point δ of the second task τ_2 there are k distinct interval structures of τ_2 , where k is dependent on periods T_1 and T_2 only. Second, since the entire system has translational symmetry with period $\frac{T_1}{k}$, it is sufficient to consider starting points of the second task between 0 and $\frac{T_1}{k}$. Also, shifting τ_2 by $0 \leq S < \frac{T_1}{k}$ is equivalent to shifting τ_1 by $-S$ or by $\frac{T_1}{k} - S$.

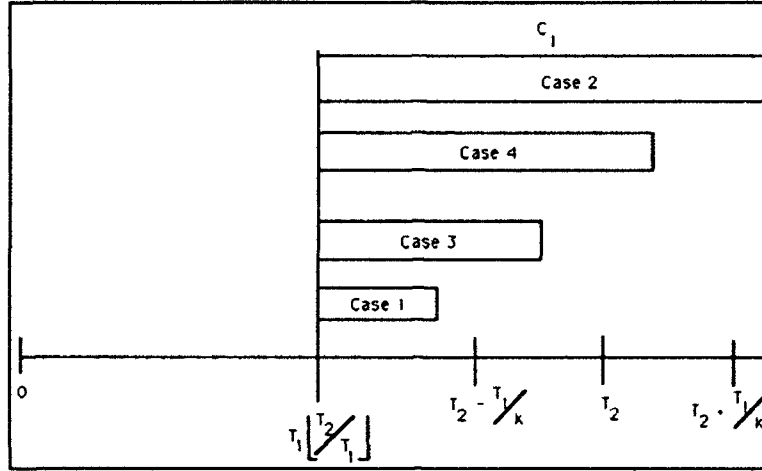


Figure 1: Relationship between C_1 and the end of the first period of τ_2

3.2 Determining Optimal Initiation Times

In this section we consider T_1 , T_2 and C_1 to be given, and we shift the starting time of τ_1 by $0 \leq S < \frac{T_1}{k}$ in order to maximize the running time of τ_2 for which the task set is still schedulable. Let C_2 be the maximum running time of τ_2 for $S = 0$ and $C_2 + C'_2$ be the maximum running time over all S .

From Lemma 1 when two tasks start together we know that the first request of the task τ_1 in I'_m arrives at $m \frac{T_1}{k}$ from the beginning I'_m , where $0 \leq m < k$. Let us consider several cases of the values of C_1 and their relationship to the starting point of the first task τ_1 (see Figure 1).

Let the second task be initiated at time 0 and the first task be initiated at time $S \leq 0$. Note also that by definition

$$|T_2 - T_1 \lfloor \frac{T_2}{T_1} \rfloor| \geq \frac{T_1}{k}$$

as long as T_2 is not a multiple of T_1 . (If T_2 is a multiple of T_1 , Liu [3] shows that a utilization of 1 can be obtained.) The time available for the execution of τ_2 in I_i is simply the time when τ_1 is not executing and it changes with S .

Due to space limitations, detailed proofs of the results shown below are not included here. However, proofs of these and other results may be found in [4].

Case 1 : When C_1 is such that the last execution of τ_1 in I_0 ends at least $\frac{T_1}{k}$ before the end of I_0 , any shift of the starting time of τ_1 to the right does not change the available time for the execution of τ_2 in the critical region, since the time gained at the beginning of the period equals the time lost at the end of it. Hence, when $0 < C_1 \leq (T_2 - T_1 \lfloor \frac{T_2}{T_1} \rfloor) - \frac{T_1}{k}$, no increase in utilization is possible.

$$U = 1 + C_1 \left[\left(\frac{1}{T_1} \right) - \left(\frac{1}{T_2} \right) \left\lceil \frac{T_2}{T_1} \right\rceil \right] \quad (2)$$

U monotonically decreases with increase in C_1 .

Case 2 : When C_1 is such that the last execution of τ_1 in I_0 ends at least $\frac{T_1}{k}$ after the end of the period of τ_2 , then $(T_2 - T_1 \lfloor \frac{T_2}{T_1} \rfloor) + \frac{T_1}{k} \leq C_1 < T_1$. Let us shift the starting point of τ_1 to the left by $0 < S \leq \frac{T_1}{k}$. The time available for execution of τ_2 in the critical region I'_1 does not change, since τ_1 will execute during the time gained at the end of the period and was executing in the time lost at the beginning of the period. Hence, as in the first case no increase in utilization is possible.

$$U = (\frac{T_1}{T_2}) \lfloor \frac{T_2}{T_1} \rfloor + C_1 [(\frac{1}{T_1}) - (\frac{1}{T_2}) \lfloor \frac{T_2}{T_1} \rfloor] \quad (3)$$

U monotonically increases with increase in C_1 .

Case 3 : When C_1 is such that the last execution of τ_1 in I_0 ends between $\frac{T_1}{k}$ before the end of I_0 and the end of I_0 , then a shift of the starting time of τ_1 to the left changes the available time for the execution of τ_2 in its first period. In this case $(T_2 - T_1 \lfloor \frac{T_2}{T_1} \rfloor) - \frac{T_1}{k} \leq C_1 \leq T_2 - T_1 \lfloor \frac{T_2}{T_1} \rfloor$

$$C'_2 = S = \frac{T_1 \lfloor \frac{T_2}{T_1} \rfloor - T_2 + \frac{T_1}{k} + C_1}{2} \quad \text{and} \quad U'_2 = \frac{C'_2}{T_2} = \frac{T_1 \lfloor \frac{T_2}{T_1} \rfloor - T_2 + \frac{T_1}{k} + C_1}{2 \cdot T_2}$$

Case 4 : When C_1 is such that the last execution of τ_1 in I_0 ends between $\frac{T_1}{k}$ after the end of I_0 and the end of I_0 then a shift of the starting time of τ_1 to the right changes the available time for the execution of τ_2 in its first period. In this case $T_2 - T_1 \lfloor \frac{T_2}{T_1} \rfloor \leq C_1 \leq T_2 - T_1 \lfloor \frac{T_2}{T_1} \rfloor + \frac{T_1}{k}$

$$C'_2 = S = \frac{T_2 - T_1 \lfloor \frac{T_2}{T_1} \rfloor + \frac{T_1}{k} - C_1}{2} \quad \text{and} \quad U'_2 = \frac{C'_2}{T_2} = \frac{T_2 - T_1 \lfloor \frac{T_2}{T_1} \rfloor + \frac{T_1}{k} - C_1}{2 \cdot T_2}$$

In both Case 3 and Case 4 the new utilization is

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C'_2}{T_2} \quad (4)$$

where C'_2 is the maximum possible increase in C_2 in the critical time zone, due to the optimal starting times of the tasks.

$$U = \frac{1}{2} + (\frac{1}{2} * \frac{T_1}{T_2} \lfloor \frac{T_2}{T_1} \rfloor) + (\frac{1}{2k} * \frac{T_1}{T_2}) + \frac{C_1}{T_2} (\frac{T_2}{T_1} - \lfloor \frac{T_2}{T_1} \rfloor - \frac{1}{2}) \quad (5)$$

Let $f = \frac{T_2}{T_1} - \lfloor \frac{T_2}{T_1} \rfloor$. Then utilization U is a function of f and has the following dependency on it.

- If $f < \frac{1}{2}$ then U monotonically decreases with increase in C_1 .
- If $f > \frac{1}{2}$ then U monotonically increases with increase in C_1 .
- If $f = \frac{1}{2}$ then U is a constant.

The figure 5 represents the variation of the processor utilization bound with the execution time of the first task C_1 . It also compares the utilization bound obtained by having both tasks start at the same time $U(\text{old})$, with that obtained by having them start at different times, $U(\text{new})$. We notice that $U(\text{new})$ is an improvement on $U(\text{old})$ in regions of C_1 where $U(\text{old})$ has its minimum value.

3.3 Modifying Task Periods with Optimal Initiation Times

In this section, we try to improve the processor utilization bound by changing slightly, the periods T_1 or T_2 or both, of the two tasks τ_1 and τ_2 , respectively. In section 3.2 C_1 , T_1 and T_2 are assumed to be constants and C_2 is adjusted to fully utilize the available processor time in the critical time zone. Then the difference in starting times of the two tasks that gives the maximum possible increase in C_2 is determined and U is given by equations 5. Here we assume that the two tasks start at the different starting times, such that it gives the best improvement of utilization bound.

In the range

$$T_2 - T_1 \lfloor \frac{T_2}{T_1} \rfloor - \frac{T_1}{k} \leq C_1 \leq T_2 - T_1 \lfloor \frac{T_2}{T_1} \rfloor + \frac{T_1}{k}$$

improvements of utilization are possible.

1. If $f < \frac{1}{2}$ U monotonically decreases with increase in C_1 . Minimum occurs at

$$C_1 = T_2 - T_1 \lfloor \frac{T_2}{T_1} \rfloor + \frac{T_1}{k}$$

The utilization bound is

$$U(\text{new}) = 1 - \frac{f(1 - (f + \frac{1}{k}))}{I + f} \quad (6)$$

where

$$f = \frac{T_2}{T_1} - \lfloor \frac{T_2}{T_1} \rfloor \text{ and } I = \lfloor \frac{T_2}{T_1} \rfloor.$$

2. If $f > \frac{1}{2}$ U monotonically increases with increase in C_1 . Minimum occurs at

$$C_1 = T_2 - T_1 \lfloor \frac{T_2}{T_1} \rfloor - \frac{T_1}{k}$$

The utilization bound is

$$U(\text{new}) = 1 - \frac{(1 - f)(f - \frac{1}{k})}{I + f} \quad (7)$$

3. If $f = \frac{1}{2}$ U is a constant. Minimum occurs over the whole range

$$T_2 - T_1 \lfloor \frac{T_2}{T_1} \rfloor - \frac{T_1}{k} \leq C_1 \leq T_2 - T_1 \lfloor \frac{T_2}{T_1} \rfloor + \frac{T_1}{k}$$

$U(\text{new})$ is given by any of the above equations 6, 7.

In the paper by Liu and Layland, the utilization bound is

$$U(\text{old}) = 1 - \frac{f(1 - f)}{I + f} \quad (8)$$

Each value of f has only one value of k associated with it. If $f = \frac{a}{b}$, where a, b are relatively prime integers, then $k = b$. Thus given the value of f , the value of k can be determined. Each finite value of k , $k > 2$, has finite number of values of f associated with it. The values of f associated with k correspond to $\frac{x}{k}$, where x is any integer, $1 \leq x < k$, and x and k are relatively prime. For example, if $k = 6$, the corresponding f values are $\frac{1}{6}$ and $\frac{5}{6}$. Thus, the number of values of f associated with a single value of k

depends on whether k is prime, and if not, the number of integers less than k that have a common factor with k . Thus for a particular k , all possible f values can be generated, and hence all possible values of the utilization bound.

Table 1 (Appendix A) shows the various utilization bounds for $2 < k \leq 20$. It is obtained by generating all possible f values for each k value and substituting in equations 6, 7.

Graph 1, obtained from this table, is a plot of both $U(old)$ and $U(new)$ for various values of f and k .

3.4 Safety Margin

In a lot of applications, task execution times are stochastic, and the worst case execution time is much more than the average execution time. If the system was designed to take care of all overloads, a very low utilization would result. Hence, in order to have better utilization the system must have the capacity to handle occasional transient overloads.

Increase in execution times of the tasks, increases the utilization of the system. If the utilization bound is high enough, this will not affect the schedulability of the system. Thus, the *safety margin*, which we define as the difference between the utilization bound of the system and the actual utilization of the system, becomes important.

In this section we compare the safety margins of two systems, defined as follows.

System 1 : A system of two tasks τ_1 and τ_2 with periods T_1 and T_2 , execution times C_1 and C_2 , respectively. Both tasks are initiated at the same time as considered in Liu and Layland's paper.

System 2 : Same as the first system, except that

- The starting time of one task is changed relative to the other, so as to give the best possible improvement to the utilization bound and
- T_2 is changed to T'_2 , by an amount z , to get further improvement in the utilization bound, as shown in the previous section.

In System 1, let U_A be the actual utilization of the system and U_B be the utilization bound of the system.

In System 2, let U'_A be the actual utilization of the system and U'_B be the utilization bound of the system. k' is the least common multiple of T_1 and T'_2 divided by T'_2 .

Equations for U_A , U_B , U'_A are obtained from [3]. Equations 2, 3, 5 are used to determine U'_B .

Now we compare the safety margins of the two systems, over $0 \leq C_1 \leq T_1$. We define the increase in Safety Margin to be

$$(U'_B - U'_A) - (U_B - U_A) \quad (9)$$

We define,

$$x = T_2 - T_1 \lfloor \frac{T_2}{T_1} \rfloor \quad \text{and} \quad y = T'_2 - T_1 \lfloor \frac{T'_2}{T_1} \rfloor$$

We assume, $\lfloor \frac{T_2}{T_1} \rfloor = \lfloor \frac{T'_2}{T_1} \rfloor$

This assumption is valid because, as shown in the previous section, k is decreased by changing T_2 in such a way as to obtain the largest possible common factors between $(T_2 \bmod T_1)$ and T_1 . All possible values of $(T_2 \bmod T_1)$ can be obtained by changing T_2 , but still maintaining $\lfloor \frac{T_2}{T_1} \rfloor = \lfloor \frac{T_2'}{T_1} \rfloor$

$$\text{Also, } z \leq \frac{T_1}{k}$$

Since the periods of the tasks are changed only to improve utilization bounds, it is reasonable to assume that small changes of periods are preferable.

The Figure 2 represents the relationship between the utilization bound and C_1 (the execution time of the first task), for both systems.

Case 1 : T_2 is increased by z . The actual system utilization decreases. Clearly for some ranges of C_1 , the utilization bound increases for system 2 and hence there is an improvement in the safety margin. In the other ranges of C_1 , equations for utilization bound are considered and the conditions under which there is an improvement in the safety margin of system 2 are derived. Refer Appendix B.

Case 2 : T_2 is decreased by z . For some ranges of C_1 , there is an improvement in the utilization bound of system 2. But since T_2 is decreased, there is an increase in the actual system utilization. Hence for all ranges of C_1 , the conditions for the improvement of the safety margin are derived from the equations for the actual utilization and the utilization bound of the two systems. Refer Appendix B.

A method to choose z

From the results obtained, we describe below, a method to choose z .

If the possibility of variation of the computation time of the first task (C_1) is known to be in the low ranges i.e., below x , then an increase in T_2 by z would give an increase in safety margin. The only restriction on z is that $z < \frac{T_1}{k}$, since this would mean that $z < \frac{T_1}{k'}$, since $k > k'$.

If it is known to be in higher ranges i.e., above x , and the utilization of the system is in the range

$$U > \frac{1}{2} - f \frac{T_1 - C_1}{T_2}$$

then an increase in safety margin can be obtained by decreasing T_2 by z , where z is chosen to satisfy the conditions,

$$z < \frac{T_1}{k} \quad \text{and} \quad z < \frac{\frac{T_1}{2}(f + \frac{1}{k}) - \frac{C_1}{2}}{\frac{C_1}{T_1} + \frac{C_2}{T_2} - \frac{1}{2} + f \frac{(T_1 - C_1)}{T_2}}$$

In both cases there is an assumption that $z < \frac{T_1}{k}$. Since we change T_2 in order to reduce k , $k' < k$ and hence, we can choose $z < \frac{T_1}{k}$. Also after z is determined, the condition $\lfloor \frac{T_2}{T_1} \rfloor = \lfloor \frac{T_2'}{T_1} \rfloor$ should be satisfied.

3.5 Using the Results

In this section we see how the results obtained in section 3.2 and section 3.3 can be applied. In section 3.2 we determined that if the two tasks start at different times, then depending on the difference in starting

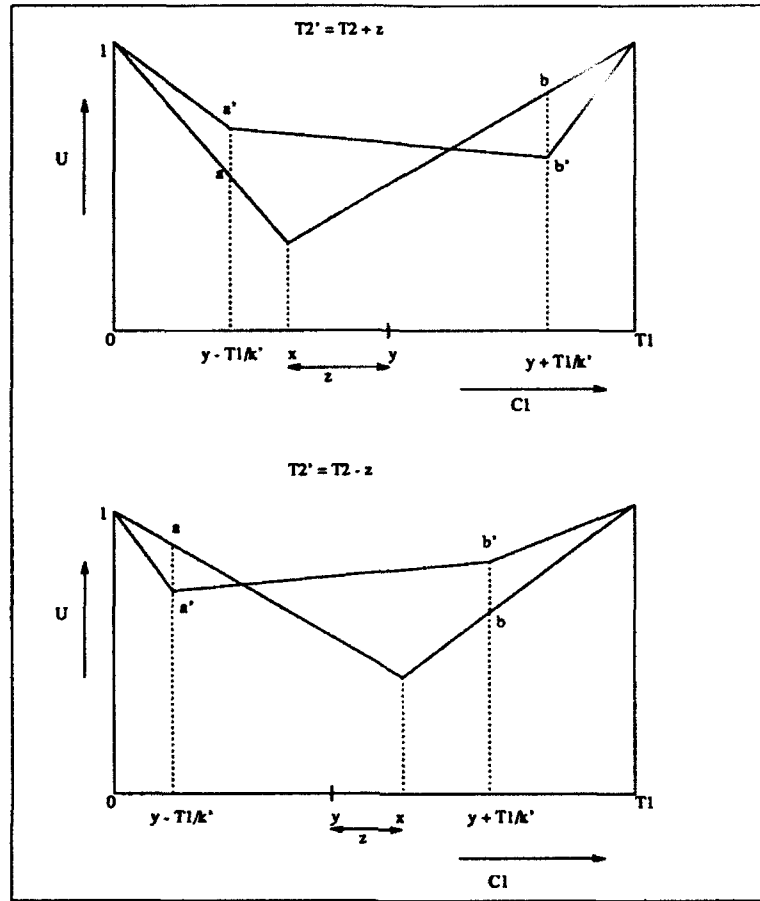


Figure 2: Variation of utilization bound with C_1

times, there is an increase in utilization factor for some ranges of C_1 , where the value of utilization is minimum. In section 3.3 the optimal difference in starting time is used. From the equations for $U(old)$ i.e., both tasks starting at the same time, and the equations of $U(new)$ i.e., with the optimal time difference between the starting points of the two tasks, we see that $U(new)$ is definitely improved by a positive value $\frac{1-f}{T+f} * \frac{1}{k}$ for $f \geq \frac{1}{2}$ and $\frac{f}{T+f} * \frac{1}{k}$ for $f \leq \frac{1}{2}$. Thus, just by having the tasks to start at different times, and especially if the difference is optimal, the upper bound of utilization factor can be improved.

Another way to improve the processor utilization bound is by changing slightly T_1 or T_2 or both. $U(old)$ is a function of f . f is a positive fraction $0 \leq f \leq 1$. The minimum value of $U(old)$ occurs for $f = 0.4$. $U(old)$ increases as the value of f moves away from 0.4 towards 0 or 1. Small changes in f cause small changes in $U(old)$. Thus, it is possible to increase $U(old)$ by changing the value of f . Since the minimum value of $U(old)$ lies at $f = .414$, the value f of should be moved away from .414.

$U(new)$ is a function of both f and k . With change in f it behaves in a similar fashion to $U(old)$ except that some values of f give lower values of k , and for such values, $U(new)$ improves drastically. The best value of k is 2, and this gives a utilization factor of 1. Thus, it is possible to increase the upper bound to utilization factor by decreasing the value of k . In particular if $k = 2$, $U(new)$ improves to 1.

However for very high values of k , say above 20, the change in $U(\text{new})$ for small change in k is small.

Small changes in T_2 , with T_1 constant, introduce small changes in f . Small changes in T_1 , with T_2 constant, introduce either small or large variations in f .

Consider T_1 being constant and T_2 being varied slightly. There could be only a small increase in $U(\text{old})$. However, this could result in a large increase in $U(\text{new})$, since changes in T_2 can reduce k to a low value.

k can be reduced in the following way. $f = \frac{T_2}{T_1} - \lfloor \frac{T_2}{T_1} \rfloor$, f can also be written as $\frac{T_2 \bmod T_1}{T_1}$. If $f = \frac{a}{b}$, a, b being relatively prime integers, then $k = b$. Thus, if T_1 is prime, or $T_2 \bmod T_1$ is prime, or T_1 and $T_2 \bmod T_1$ are relatively prime then $k = T_1$ will be high since $1 < k \leq T_1$. Otherwise $f = (\frac{T_2 \bmod T_1}{T_1})$ can be rewritten as $f = \frac{a}{b}$ where a, b, i are integers such that $T_2 \bmod T_1 = a \cdot i$, and $T_1 = b \cdot i$ and $i \neq 1$, so $k = b$, $b < T_1$ and so k is small. The closer i is to $T_2 \bmod T_1$ the better.

EXAMPLE

Here we present a numerical example which illustrates the gains possible by modifying initiation times and periods. Consider the case when T_1 , not a prime number is held constant and T_2 can be varied slightly. T_2 can be adjusted so that $T_2 \bmod T_1$ has a common factor with T_1 , as close to $T_2 \bmod T_1$ as possible. Let $T_1 = 18$.

1. If $T_2 = 26$ then $T_2 \bmod T_1 = 8, GCD = 2$.

$$f = \frac{T_2 \bmod T_1}{T_1} = \frac{8}{18} = \frac{4}{9}$$

$$k = 9$$

$$U(\text{old}) = .829$$

$$U(\text{new}) = .867$$

2. If $T_2 = 27$ then $T_2 \bmod T_1 = 9, GCD = 9$.

$$f = \frac{T_2 \bmod T_1}{T_1} = \frac{9}{18} = \frac{1}{2}$$

$$k = 2$$

$$U(\text{old}) = .833$$

$$U(\text{new}) = 1.0$$

3. If $T_2 = 28$ then $T_2 \bmod T_1 = 10, GCD = 2$.

$$f = \frac{T_2 \bmod T_1}{T_1} = \frac{10}{18} = \frac{5}{9}$$

$$k = 9$$

$$U(\text{old}) = .841$$

$$U(\text{new}) = .873$$

From the above three cases it is evident that if the original value of T_2 was either 26 or 28, then by just changing it to 27 the utilization bound can be improved from .867 to 1.0 and from .873 to 1.0, respectively.

When T_2 is changed to 27 from 28 however, there is an increase in the actual utilization. If for instance $C_1 = 5$ and $C_2 = 8$, the actual utilization increases from .564 when $T_2 = 28$ to .574 when

$T_2 = 27$. But this increase is small when compared to the improvement in utilization bound. Also it is not always necessary to decrease T_2 , an increase in T_2 could also improve the utilization bound while decreasing the actual utilization. In either case, the safety margin, which is the difference between the actual utilization and the bound, may be increased considerably.

Keeping T_2 constant and changing T_1 could also similarly improve $U(new)$ by reducing k sufficiently.

However it should also be noted that, since $U(new)$ is a function of both k and f , in some cases a reduction in the value of k alone is not sufficient to guarantee an increase in $U(new)$. Usually the best improvement can be obtained when the original value of f is close to .4. If the original value of f is close to 0 or 1, the utilization bound is already high and only when changes in T_1 or T_2 reduces k to 2, an increase is guaranteed because $U(new)$ improves to 1.

Also, this method of having different starting points, to improve the upper bound to utilization can only be used if the ratio of the periods of the two tasks is rational. This question however does not arise in practice because both periods would be derived from the system clock, and hence, their ratio will always be rational.

4 Many-Task Systems

4.1 Introduction

When we consider a general n task system, an analytical approach to determine the optimal initiation times for the tasks, by avoiding critical instants, in order to improve the utilization bound is extremely complex. This is because of the dependance of the structure of the lower priority periods on higher ones. Here we describe an algorithm to determine the initiation times for the task set, which results in a better utilization bound and hence a better safety margin. This algorithm favors tasks with a higher priority i.e., tasks with a lower period. In [5] the period transformation method is described, by which the periods of the more important tasks is transformed to values smaller than periods of less important tasks or vice versa. This makes the *priority* of the tasks equal to its *criticality*. This idea could be used to enhance the performance of our algorithm. We also describe a method to reduce the periods slightly and obtain a better utilization bound.

In [1], a stochastic analysis of the performance of the rate monotone algorithm is presented. A task set is generated randomly and the computation times are multiplied by a small factor δ and δ is systematically increased to a threshold value at which some task deadline is missed. The utilization corresponding to this value of δ is defined as the *breakdown utilization*. We evaluate the performance of our algorithms by determining the breakdown utilization of the task set whose initiation times are determined or whose periods are changed by our algorithms, and comparing it with the breakdown utilization obtained with the original specifications. It must be noted that at the point of breakdown the system may not be *fully utilized*. It may still be possible to increase the computation times of some of the tasks and still keep the system schedulable. However evaluating the system by determining the breakdown utilization is a more realistic approach.

4.2 Determining Initiation Times

4.2.1 - An Algorithm to determine Initiation Times

The system consists of n periodic tasks $\tau_1, \tau_2, \dots, \tau_n$ with periods T_1, T_2, \dots, T_n and execution times C_1, C_2, \dots, C_n , respectively. Without loss of generality it is assumed that $T_1 < T_2 < \dots < T_n$, and also that T_1, T_2, \dots, T_n are all integers. The task with a smaller period has a greater priority and the tasks are scheduled by the rate monotone scheduling algorithm. The *criticality* of a task is assumed to be the same as the priority given to the task.

The algorithm described below improves the safety margin of the system by initiating each task at a specific time. The initiation time for each task τ_i is determined, taking into account all the tasks with a higher priority, because by the rate monotone scheduling algorithm the schedulability of a particular task is affected only by tasks of higher priority. The optimal initiation time of a task τ_i maximizes the available time for the execution of the task, assuming that the tasks $\tau_1, \tau_2, \dots, \tau_{i-1}$ are initiated at their respective optimal initiation times. The algorithm fits tasks into the schedule one by one, starting with the task with the highest priority and going down to the last task in the order of decreasing priority.

1. The greatest common divisor of T_1, T_2, \dots, T_n and C_1, C_2, \dots, C_n is determined to be gcd .
2. The task τ_1 is initiated at $t = 0$.
3. Each subsequent task is initiated at times $t \geq 0$ to determine the optimal initiation time for the task. We do not consider initiation times $t < 0$ due to symmetry.
4. The initiation time for task i is determined in step i .
5. s_i denotes the initiation time for task i and S_i denotes the optimal initiation time for task i .
6. LCM_i denotes the least common multiple of T_1, T_2, \dots, T_i .
7. $C_{i_{max}}$ denotes the maximum possible value of C_i for which the task τ_i remains schedulable, assuming $\tau_1, \tau_2, \dots, \tau_{i-1}$ are initiated at S_1, S_2, \dots, S_{i-1} respectively and τ_i at s_i .
8. δ_{i-1} denotes the smallest possible *busy CPU block* or *free CPU block* at the beginning of step i when s_i is a multiple of δ_{i-1} .

Consider step i i.e., tasks $\tau_1, \tau_2, \dots, \tau_{i-1}$ are initiated at S_1, S_2, \dots, S_{i-1} respectively and S_i is being determined. We do not consider C_i since $C_{i_{max}}$ is to be determined.

Consider the region $t = s$ to $t = LCM_i + s$, where s is the initiation time of τ_i . This can be divided into k regions R_1, R_2, \dots, R_k , each of size T_i .

Free CPU blocks and busy CPU blocks (see Figure 3) are defined within each region R_x where $x = 1, 2, \dots, k$. A *free CPU block* is a continuous region within a region R_x where the CPU is idle. A *busy CPU block* is a continuous region within a region R_x where the CPU is busy.

In step 1, $\delta_1 = gcd$ and $S_1 = 0$. In each step i , i ranging from 2 to n ,

1. $\delta_i = \delta_{i-1}/2$
2. s_i is varied from 0 in steps of δ_i upto the greatest common divisor of T_i and LCM_{i-1} . For each s_i , $C_{i_{max}}$ is determined. The value of s_i that maximizes $C_{i_{max}}$ is the optimal value S_i .

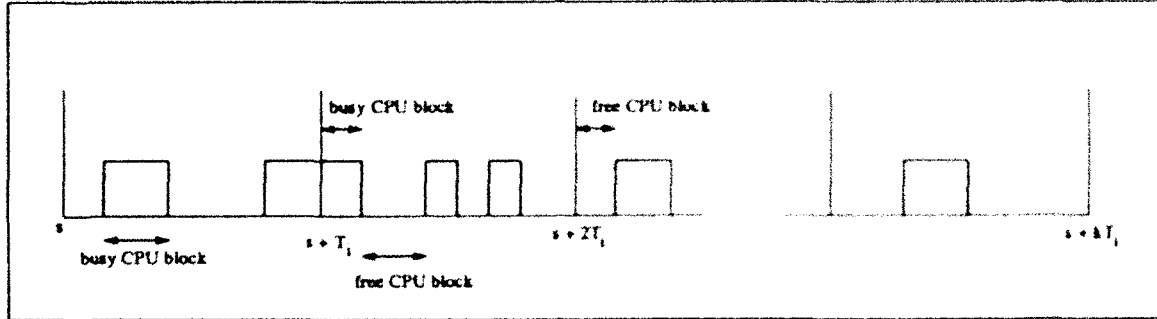


Figure 3: Free and Busy CPU Blocks

We do not concern ourselves with the computational complexity of this algorithm right now because the rate monotone schedule is designed off-line.

4.2.2 Effectiveness of the algorithm

Our algorithm does not always determine optimal initiation times. However, we present here some observations about the structure of the periods of the task system, which suggest that the results obtained using our algorithm will be close to optimal.

1. The region between $t = s_i$ (s_i , a multiple of δ_{i-1}) and $t = LCM_i$ can be divided into k regions R_x , $x = 1, 2, \dots, k$ each of size T_i . We define these regions to be k -regions.

The difference between the time available for the execution of τ_i in any two regions R_x is a multiple of δ_{i-1} .

This is because any free or busy CPU block in any region R_x is a multiple of δ_{i-1} .

2. Starting from 0 or any multiple of δ_{i-1} , if we move the initiation time of τ_i by any distance x less than or equal to δ_{i-1} , then the change in available time for execution of τ_i in any region either
 1. increases exactly by x
 2. decreases exactly by x
 3. remains the same.

The only extreme cases to be considered are shown in Figure 4.

Over all the regions between s_i and $LCM_i + s_i$, the total gain in time for execution of τ_i equals total loss.

3. From the above two observations hence, maxima for the time available for the execution of τ_i , will be obtained only at s_i values that are multiples of $\delta_i = \frac{\delta_{i-1}}{2}$.

In the extreme case, two regions R_k and R_j differ by δ_{i-1} , the time available for the execution of τ_i in R_k is x and in R_j is $x + \delta_{i-1}$. Also x is the minimum time available for the execution of τ_i over

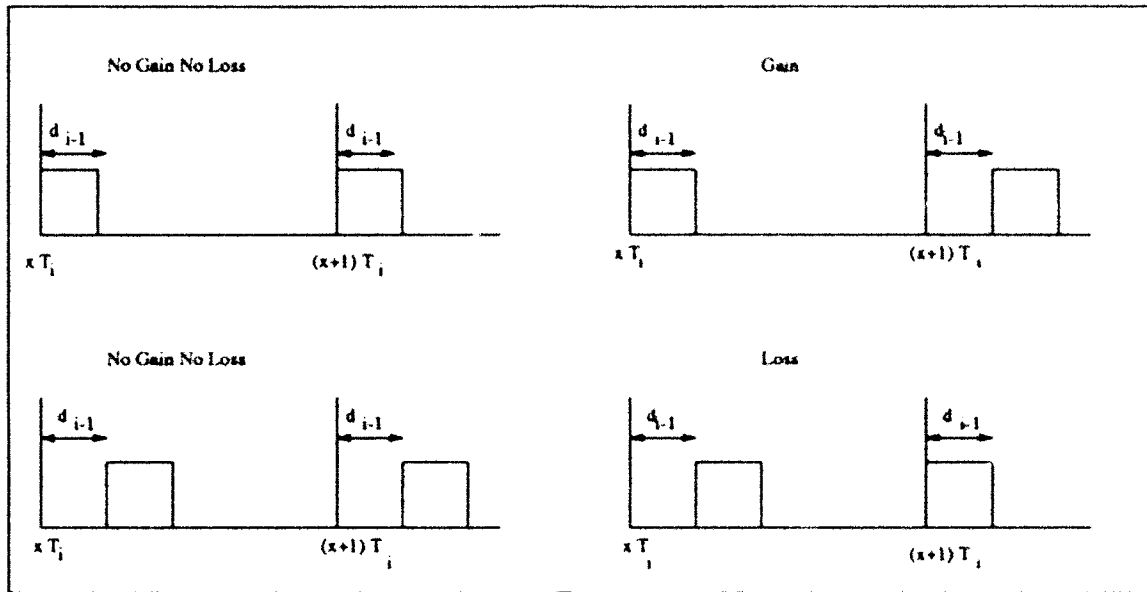


Figure 4: Extreme Cases

all the regions R_1, R_2, \dots, R_k . Now if the initiation time is moved by $\delta_i = \frac{d_{i-1}}{2}$ and time available for the execution of τ_i in R_k is $x + \delta_i$ and R_j is $x + \delta_i$ and this is a maximum. Hence if all values for s_i that are multiples of δ_i are considered, a maximum value for the time available for the execution time of τ_i will not be missed.

4.3 An Algorithm to determine the Periods

From the analysis of a two task system we determined that the utilization bound of the system improves when the k value (k is LCM of T_1 and T_2 divided by T_2) is reduced, or in other words the greatest common divisor of the periods is increased, especially when the value of f is close to 0.414 (f is $\frac{T_2}{T_1} - \lfloor \frac{T_2}{T_1} \rfloor$). Our algorithm for a general n task system is based on this idea.

We assume that it is allowable to reduce the periods of the tasks by a specified percentage (say x). This is a reasonable assumption because if you consider a common real-time application such as data sampling, a faster rate of sampling is often desirable.

The tasks in the set are $\tau_1, \tau_2, \dots, \tau_n$, with periods T_1, T_2, \dots, T_n respectively with $T_1 < T_2 < \dots < T_n$. The new periods are determined by an exhaustive search over the range $.0x * T_i$ to T_i , for each task i , $1 \leq i \leq n$.

1. τ_1 and τ_2 are considered, and a combination of periods that gives the smallest k value is chosen. Let them be T'_1 and T'_2 respectively.
2. The least common multiple of T'_1 and T'_2 is determined. Let it be LCM_2 .
3. For $3 \leq i \leq n$,

T'_i is chosen so as to obtain the smallest k value or largest greatest common divisor between T'_i and LCM_{i-1} , where LCM_i is the least common multiple of T'_1, T'_2, \dots, T'_i . Also T'_i closer to T_i is favoured.

The new LCM_i is determined.

4.4 Simulation and Results

Random task sets are generated with each task set having i) two tasks each ii) three tasks each iii) five tasks each. For each of the above, the average breakdown utilizations are determined over 50 task sets.

The task sets are generated by selecting relative periods and computation times uniformly distributed, and then scaling the computation time to give a schedulable task set. The breakdown utilization is obtained by scaling the computation time systematically till it becomes unschedulable.

4.4.1 Improvements due to change in Task Initiation times

The initiation times for the tasks are determined and a comparison of the breakdown utilization is made with the task system that has all tasks initiated together. The results are tabulated in Table 1.

number of tasks in task set	breakdown util 0 init time	breakdown util with init time	difference in breakdown u	number of tasks that improve
Periods between 2 and 20				
2	0.943173	0.951178	0.008005	38
3	0.913220	0.922900	0.009300	48
5	0.893316	0.909797	0.164810	50
Periods between 2 and 50				
2	0.948143	0.948415	0.000272	28
3	0.920709	0.922934	0.022250	38
Periods between 2 and 100 multiples of 5				
2	0.944266	0.954509	0.0102430	N/A
3	0.914785	0.927008	0.012223	N/A
4	0.884467	0.904569	0.020102	N/A

Table 1

The results indicate the following trends

- There is a definite improvement in the average breakdown utilization of the task sets when the initiation times of the tasks is changed. This is because for each task the critical instant is avoided, each of the k -regions is considered and the initiation time is so chosen as to maximize the minimum time available for the task. Hence each task has a little more computation time available and so breakdown utilization increases.
- As the number of tasks in the task set increases, there is an increase in the improvement of the utilization bound. This is an interesting observation and it suggests that as the size of the task

set increases a lot more can be gained by shifting the initiation times. This is because the time available for the execution of each task is maximised, each task gains a little and when the number of tasks increases the effect is cumulative.

- We also observe that the extent of the improvement is very small (an average of .5 percent). However these are average values and if individual task sets are considered, the actual improvement in breakdown utilization could go up to 6 to 8 percent.
- The above results also indicate that unschedulable task sets can be made schedulable and Table 2 has some examples.

task	period	computation	tasks initiated together	tasks initiated at specific times	
priority		time	max comp time	init time	max comp time
Two task system					
1	100	45	100	0.0	100.0
2	250	120 unschedulable	115	22.5	137.5
Three task system					
1	120	30	120	0.0	120.0
2	150	65	90	15.0	105.0
3	200	30 unschedulable	25	112.5	57.5
Five task system					
1	100	20	100	0.0	100.0
2	150	45	110	10.0	120.0
3	350	50	150	162.5	150.0
4	400	110 unschedulable	100	45.0	122.5
5	720	20		115.0	27.5

Table 2

The only change that would have to be made is in the initiation time of the tasks. Therefore there is no change in the actual utilization of the task set. However we see an improvement in the breakdown utilization and hence an improvement in the safety margin of the system, at practically no cost.

4.4.2 Improvements due to change in Task Periods and Initiation Times

As the range of the periods decreases the improvement in average breakdown utilization obtained by shifting the initiation time increases. Refer 4.4.1.

Also if the periods are harmonic, which is the case in a lot of real-time applications, then the improvement in the breakdown utilization is better. Refer 4.4.1.

The periods of the tasks are changed slightly (allowed a reduction up to 10 percent) and then the initiation times are determined. The breakdown utilization is determined and the results are compared with 1. The original system 2. The system with change in periods only, with all tasks initiated together.

The results are tabulated in Table 3.

number of tasks in task set	actual util	breakdown util	safety margin	improvement in safety margin
All tasks initiated together				
2	0.699809	0.940779	0.240970	-
3	0.699616	0.912586	0.212926	-
5	0.699276	0.871999	0.172723	-
Periods changed and all tasks initiated together				
2	0.727403	0.953687	0.229657	-0.011313
3	0.732088	0.930850	0.198762	-0.014164
5	0.737715	0.888321	0.150606	-0.022117
Periods changed and tasks initiated at specific times				
2	0.727403	0.978309	0.259060	0.180900
3	0.732088	0.953566	0.221478	0.008552
5	0.737715	0.931120	0.193406	0.020737

Table 3

The results indicate that

- There is a substantial improvement in the breakdown utilization of the task set. Since task periods are reduced, there is also an increase in the actual utilization of the system, however, this is less than the improvement obtained in the breakdown utilization and hence there is an overall improvement of about 1 percent in the safety margin. For an n task system a utilization bound of 1 can be obtained by making the period of the n^{th} task a multiple of all other task periods [3]. This means a k value of 1. For each task we maximize the bound by decreasing the k value and providing additional gain by changing initiation times.
- We also observe that if the periods alone are changed, and the tasks all initiated together there is a small improvement in the breakdown utilization but because of the increase in actual utilization there is a decrease in the safety margin.
- It should be noted here also that these are average values and if specific task sets are considered the improvements in safety margin are more substantial.

Thus an improvement in the safety margin of the overall system at no run-time cost. In fact there may also be improvement in application performance if the periods of sampling, monitoring etc., are reduced.

5 Conclusion

We have presented some enhancements to the rate monotone algorithm which exploit application characteristics to avoid worst case situations and improve the safety margin of critical real-time systems without run-time penalty.

We have determined the optimal initiation times for a 2-task system using the rate monotonic scheduling algorithm, and the resulting increase in the utilization bound. It is only possible to improve the bound in some cases, however sometimes significant improvements may be obtained. In addition, the extent of improvement in the bound depends on the relationship of the periods of the two tasks. Contrary to intuition, sometimes reducing task periods may sometimes substantially increase the bound, and transform a previously infeasible task set into a feasible one. By changing the periods and using the optimal initiation times an increase in the safety margin is also obtained.

The techniques used for the two-task case, which involve analyzing the structure of task arrival patterns, and determining the relationship between periods, increase quickly in complexity as the number of tasks increases. So algorithms were designed for determining initiation times and periods, and improvements in safety margin were obtained. Also as the size of the task set increases the gains due to modified initiation times increases. Reducing periods also sometimes produces a gain in the safety margin.

Results are sufficiently promising so that a system designer may wish to put in the effort to select the best initiation times and task periods, to enhance the schedulability and safety of the system.

6 Acknowledgements

We would like to thank C. L. Liu, Lui Sha, Wei Zhao, Richard Volz and Dong-Won Park for many valuable comments and suggestions through out this work.

References

- [1] John Lehoczky, Lui Sha, and Ye Ding. The rate monotone scheduling algorithm: Exact characterization and average case behaviour. Technical report, Department of Statistics, Carnegie Mellon Univ. Pittsburgh, 1987.
- [2] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in hard-real-time environment. *Proc. IEEE Real-Time Systems Symposium*, pages 261-270, 1987.
- [3] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 20:46-61, 1973.
- [4] R. Menon, M. J. Kim, A. Kanevsky, and S. Natarajan. Improving safety margins in rate-monotone scheduling. Technical Report TAMU-CS-92-014, Texas A&M University, 1992.
- [5] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized pre-emptive scheduling. In *Proc. IEEE Real-Time Systems Symposium*, pages 181-189, 1986.
- [6] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. Technical Report 181, Dep. Computer Sc., Carnegie Mellon Univ. Pittsburgh, 1987.
- [7] B. Sprunt, J.P. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Proc. IEEE Real-Time Systems Symposium*, 1988.
- [8] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard-real-time systems. *The Journal of Real-Time-Systems*, pages 27-60, 1989.

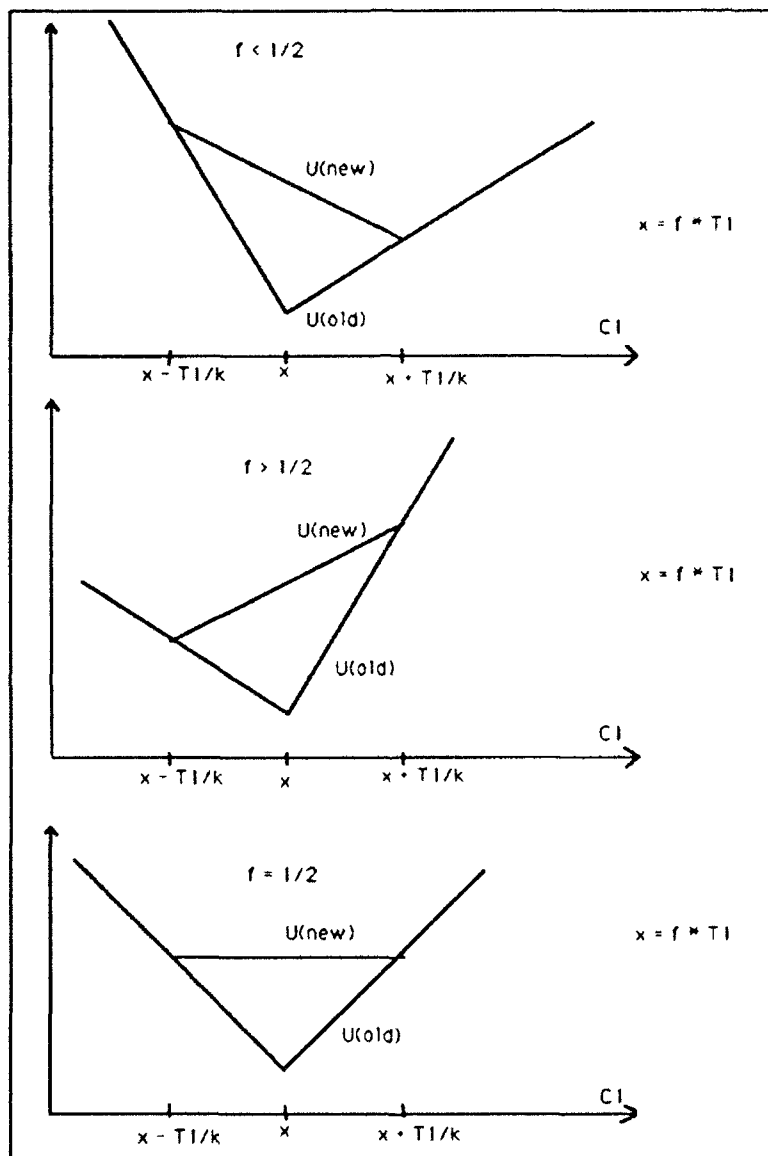


Figure 5: U as a function of C_1

Stochastic Scheduling for Distributed Real-Time Systems*

Hossein Moiin and P.M. Melliar-Smith
Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106

Abstract

This paper presents a novel strategy for scheduling tasks in a distributed real-time system. The system is composed of several processors that communicate via dedicated links. Tasks in the system are either periodic or aperiodic. The schedule for each processor of the system must be constructed dynamically as aperiodic tasks arrive at unpredictable times. Each processor of the distributed system is capable of executing any of the aperiodic tasks, while the periodic tasks are executed locally. The scheduling strategy is divided into two components, a local scheduling strategy responsible for timely execution of tasks arriving at a processor and a global strategy responsible for the selection and stochastic transfer of those tasks that cannot be executed locally. The global scheduler uses a stochastic and learning algorithm to reduce the number of late tasks. The simulations identify the circumstances in which stochastic scheduling is superior to deterministic scheduling.

I Introduction

Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. These systems can be characterized by presence of tasks that have timing requirements, such as deadlines. Examples of real-time systems include, process control systems, aircraft flight control systems, nuclear power plant safety systems and air traffic control systems. Such systems must perform certain actions in a timely manner and their failure to do so may result in severe consequences. A number of such systems, such as nuclear power plant control systems and local area networks controlling the operation of an aircraft carrier, are composed of processes that are inherently distributed, suggesting the possibility of using a distributed system for their implementation. The number of practical systems implemented as distributed systems has, however, been limited by the difficulty of scheduling real-time tasks in a distributed system.

The scheduling of real-time tasks for a distributed system has received considerable attention in recent years (see [1] for a survey). Most of the proposed strategies are, however, aimed at environments where tasks are fully characterized in advance and are suitable only for applications that operate in a static environment.

*Partially supported by National Science Foundation grant number: NCR-9016361

In contrast, a dynamic scheduling strategies allows the system to handle tasks that are unexpected and occur at unpredictable times [6, 8, 9]. This additional flexibility increases the system's ability to adapt to external events and permits for exception handling, but also reduces system's predictability. A possible compromise is to guarantee timely execution of periodic tasks known at system initialization time and then apply a dynamic scheduling strategy to tasks that arise at unpredictable times. This compromise incorporates the flexibility of a dynamic scheduling strategy and the predictability of a static scheduling strategy. The stochastic-learning scheduling strategy uses such a compromise in a loosely coupled network.

The stochastic-learning scheduler uses a two-level scheduling strategy for scheduling of real-time tasks in a distributed system. A real-time task that enters the system through a given processor is first processed by the local scheduler at that processor. If the local scheduler cannot meet the timing requirements of a task the global scheduling algorithm selects another processor of the distributed system and transfers the task to that processor for remote execution. The local scheduler of the remote processor attempts to fit this remote task into its existing schedule so that its timing requirements are met. The global scheduler sends and receives information about its past decisions and uses this information to make better decisions in the future.

Real-time systems are often used to control *critical* applications but may experience brief periods during which some processors of the system are overloaded. The real-time control system must minimize the number of tasks that miss their timing requirements during overload situations. While a simple local scheduling strategy is sufficient for scheduling of tasks in non-overload situations, a global scheduling strategy is required to handle situations where the demand on some processors of the distributed system is higher than their capacity. Under such circumstances, the real-time system should continue to execute tasks that satisfy their timing requirements and should minimize the number of tasks that miss their deadlines. This paper uses a stochastic-learning scheduling algorithm to reduce the number of tasks that miss their deadlines. While many metrics are important in evaluating a real-time system, the most important measure for such an evaluation is the proportion of tasks that miss their timing requirements. This paper evaluates the stochastic-learning scheduling algorithm with respect to this metric and compares its performance with several other algorithms.

The rest of this paper is organized as follows. Section 2 describes the model of the distributed system and the tasks in the system. This section also outlines the overall structure of the scheduler. Section 3 is a description of the local and global scheduling strategies and of the optimal guarantee procedure. In this section data structures used by the local and global schedulers are also described. Section 4 summarizes the simulation results identifying circumstances under which the stochastic-learning global scheduling strategy is superior to other global scheduling strategies. Some concluding remarks and directions for future research in this area end the paper in Section 5.

II System Description

The system under consideration is a distributed real-time system composed of several homogeneous processors that communicate by exchanging messages on unidirectional dedicated links. Both processors and

links are assumed to be fault free and any processor of the distributed system can execute any of the aperiodic real-time tasks. Furthermore, the local clocks of all processors in the system are assumed to be synchronized and only real-time tasks are assumed to be present in the system. A real-time task is defined as a task that must complete its execution prior to its assigned deadline.

II-A Task Model

There are two types of real-time tasks in the system, *periodic* and *aperiodic*. A *periodic* task consists of a computation that is executed repeatedly, once in each fixed period of time. An example of a periodic task is reading of a sensor or generating a control output. An *aperiodic* task consists of a computation that responds to internal or external events. This type of task occurs in the system just once and at unpredictable times. Typical usage of such tasks includes responding to operator requests and exception handling. Since the periodic tasks are known in advance they can be guaranteed timely execution on their local processor. The aperiodic tasks, on the other hand, are not known *a priori* and therefore, may or may not be guaranteed timely execution in the processor in which they originate. It may be necessary to execute some of these tasks at a different processor to meet their timing requirements.

A periodic task, T_i , is characterized by a pair: $\{C_i, P_i\}$, representing its computation time and its period, respectively. A periodic task must be executed exactly once during each period. It is not important when the task is executed during its period. In a real-time system there are a number of such tasks each having its own period. The deadline of the j th instance of a periodic task, T_i can be calculated by $j \times P_i$. This value also represents the time at which the $j + 1$ st instance of the periodic task, T_i is ready to be executed. An aperiodic task, T_k , arrives into the system at unpredictable times and is characterized upon its arrival by its deadlines, D_k and its computation time, C_k . Such a task can be scheduled for execution any time after its arrival. Both periodic and aperiodic tasks are assumed to be preemptable. It is also assumed that the set of periodic tasks, their characteristics and their assignment to various processors of the system is known at the system initialization time. The characteristics of an aperiodic task becomes known only when it arrives at a processor. Furthermore, each processor has sufficient processing power to guarantee timely execution of the periodic tasks assigned to that processor. This assumption is enforced by the local scheduler of each processor which during the initialization phase, checks that the set of periodic tasks is schedulable. The following lemma establishes the necessary and sufficient conditions for schedulability [7] of a set of preemptable periodic tasks. It states that a set of preemptable periodic tasks is schedulable as long as the processor is not overloaded.

Lemma 1 Let $\tau_p = \{T_1, T_2, \dots, T_n\}$ be a set of preemptable periodic tasks. τ_p is schedulable if and only if:

$$\sum_{i=1}^n \frac{C_i}{P_i} \leq 1.$$

No assumptions are made regarding the arrival rate, the minimum inter-arrival time, the computation time or the deadline of an aperiodic task. However, the simulation model implies that tasks are independent

and there are no precedence constraint among them. Moreover, it is assumed that the only resource shared between the tasks is processing time.

The characteristics of tasks that are guaranteed for execution on a processor are kept in the Node Task Table (NTT), which contains the deadline, remaining computation time, earliest start time and the identification number of each guaranteed task. The characteristics of the periodic tasks is kept in the processor's Periodic Task Table (PTT). The PTT is used to extend the NTT whenever necessary.

II-B Overall Structure of the Scheduler

The scheduling strategy considered for this study consists of two components, a local scheduler and a global scheduler. A central idea used in both components is the notion of a *guaranteed* task. A task is said to be *guaranteed* by a processor of the distributed system if the task runs to completion prior to its deadline on that processor and does not cause any of the previously guaranteed tasks to miss their timing requirements. The local scheduler attempts to guarantee the timely execution of a task by executing it locally. The global scheduler attempts to guarantee the timely execution of a task by executing that task on a remote processor. Periodic tasks are guaranteed to meet their timing requirements and are executed locally. Aperiodic tasks that arrive into the system wait until they are processed by the local scheduler. The local scheduler attempts to guarantee aperiodic tasks by examining its current load. When a task cannot be guaranteed, that task is reconsidered by the global scheduler which probabilistically selects another processor for the task and forwards the task to that processor.

Once the global scheduler has sent a task to a remote processor, the local scheduler at that processor must determine if the task can be executed in a timely fashion. Tasks that arrive as a result of a global scheduling decision are treated as if they were newly arrived tasks except that they are labeled as *remote* tasks. If a remote task cannot be guaranteed to meet its timing requirements then that task is discarded and removed from the system. The decision to discard such a task is based on the observation that the processor deemed best capable of guaranteeing its timely execution was unable to do so. Furthermore, some time has been spent transferring the task to the remote processor. Therefore, the probability that some other processor is able to guarantee the timely execution of this task is quite small. Sending unguaranteed tasks to a second remote processor generates more traffic in the system and increases the length of the communication queues. Moreover, sending tasks to a second remote processor decreases the available processing time by increasing the processing required for scheduling message handling and system tasks without corresponding benefits. Simulation results have shown that sending tasks to a second remote processor reduces the system's performance and, therefore, this idea was not investigated further.

III The Scheduling Strategy

The scheduling of real-time tasks in a distributed system can be viewed as a two-level scheduling activity. At the lower level the local scheduler at each processor attempts to guarantee the timely execution of

a real-time task by calling the guarantee procedure. The guarantee procedure considers the processor's current workload and the future instances of all periodic tasks to determine whether the processor has sufficient processing power to guarantee the timely execution of the task under consideration. At the higher level the global scheduler is responsible for finding the most likely processor that can guarantee the timely execution of a locally unguaranteed task.

III-A Local Scheduler

The local scheduler itself is implemented as a periodic task. To reduce the number of tasks that miss their deadlines, the local scheduler is also invoked upon arrival of a new task if doing so will not cause any of the previously guaranteed tasks to miss their deadlines. When the local scheduler is invoked it considers in order each of the tasks that have arrived at the processor since its last invocation. The local scheduler calls the guarantee procedure once for each task to determine whether that task can be guaranteed locally. If a task is not guaranteed locally then that task is handled by the global scheduler. Once a task is guaranteed for execution on a processor its computation time, its deadline, its earliest start time and its identification number are entered into the appropriate row of NTT.

The local scheduler uses the earliest-deadline-first algorithm to schedule tasks [3]. A task with an earlier deadline will be scheduled to run before a task with a later deadline *if* both tasks are ready to be executed. This algorithm has been shown to be optimal [2] for a single processor and will find a *feasible* schedule if one exists. A schedule is considered *feasible* if all real-time tasks are able to meet their timing requirements.

The NTT is implemented as two ordered lists. The first list is a ready list which contains all the tasks that are ready to be executed. Tasks on the ready list have start times that are smaller than or equal to the current time and are ordered in the order of increasing deadlines. The second list is a waiting list of periodic tasks that are not yet ready to be executed, but are guaranteed timely execution. Future instances of periodic tasks must be considered by the guarantee procedure when it guarantees an aperiodic task. Tasks on this second list are ordered in the order of increasing start times. A task that moves from this list into the ready list is inserted into the ready list according to its deadline. Upon completion of the currently executing task that task is removed from the NTT and the dispatcher in the processor is invoked to select the top item on the ready list for execution.

All non-system tasks are assumed to be preemptable. System tasks, such as the local scheduler or the global scheduler, are considered to be more critical and hence, are not preemptable. When a task is preempted that task will be placed back on the ready list according to its deadline and its computation time is updated to reflect the processing time it has received so far. Preemptions can be planned or unplanned. A planned preemption is the result of a periodic task with an earlier deadline than the currently executing task entering the ready list. Such preemptions cannot cause any of the guaranteed tasks to miss their deadlines since they are accounted for by the local scheduler. Unplanned preemptions may occur as a result of a task or a message entering a processor. Such preemptions are allowed if and only if they will not

cause any of the previously guaranteed tasks to violate their timing requirements. Unplanned preemptions can only invoke a system task, whose characteristics are, of course, known in advance.

III-B The Guarantee Procedure

The guarantee procedure developed for this study is an optimal procedure in the sense that it will guarantee a task *if and only if* that task will not cause any of the previously guaranteed tasks to miss their deadlines *and if* there is sufficient time left to satisfy the timing requirement of this task. This procedure is called by the local scheduler once for each aperiodic task that arrives at a processor of the distributed system. The guarantee procedure decides whether each task can be guaranteed to receive enough processing time to complete its execution prior to its assigned deadlines.

The guarantee procedure operates by placing a task temporarily in the NTT and checking that all previously guaranteed tasks still meet their deadlines. Furthermore, since the periodic tasks are guaranteed to complete prior to their deadlines, it may be necessary to consider periodic tasks that are not yet ready to execute. The guarantee procedure achieves this by extending its current *window* of scheduling. The time interval from the earliest start time to the latest deadline of all the tasks in the NTT is defined as the current *window* of scheduling. The current window of scheduling can be extended by either the local scheduler or by arrival of an aperiodic task with a deadline later than the latest deadline of all tasks in the NTT. In both cases the current window of scheduling is extended by an integral multiple of the least common multiple of the periods of all the periodic tasks in the PTT.

The local scheduler may decide to extend the window of scheduling because the time is close to the start time of the next instance of a periodic task and that task is not in the NTT. The local scheduler will then extend the current window of scheduling by appending periodic tasks to the NTT. If the current window of scheduling is extended by the arrival of an aperiodic task, then deadline of the aperiodic task must fall within the newly extended window of scheduling, ensuring that all periodic tasks possibly affected by the arrival of this aperiodic task are accounted for.

After the NTT is extended to account for all periodic tasks that may be affected by the aperiodic task under consideration, we must consider whether the aperiodic task can also be guaranteed. The necessary and sufficient condition for schedulability for a set of preemptable tasks is established in the following lemma.

Lemma 2 Let $\tau_p = \{T_1, T_2, \dots, T_n\}$ be a set of preemptable tasks, where $T_i = (C_i, S_i, D_i)$. C_i is the computation time, S_i the earliest start time and D_i the deadline of T_i . Let τ_p be sorted in non-decreasing order of start times (i.e. for any pair of tasks T_i and T_j if $i > j$, then $S_i \geq S_j$). τ_p is schedulable if and only if:

$$\forall i \forall j \left(\sum_{k=i}^j C_k \right) \leq D_j - S_i.$$

The lemma requires that for each task in the set τ_p , the guarantee procedure ensures that there is sufficient time to process the task and all guaranteed tasks that are in τ_p and start later. To implement this scheduling condition requires time proportional to n^2 where n is the number of the tasks in the NTT. This implementation would be too costly in a dynamic real-time system. However, an incremental version of the above schedulability condition can be implemented in time proportional to n . This incremental version relies on two observations. First, the start time of the task under consideration is equal to the current time because only aperiodic tasks can cause a call to the guarantee procedure. Second, it is possible to construct a feasible schedule for a set of periodic tasks that are not yet ready to execute. This observation is based on a schedulability test that is performed during the system initialization phase for a set of preemptable periodic tasks. These observations can be incorporated to change the schedulability condition for a set of preemptable tasks where the schedule is constructed incrementally by including one additional task at a time.

Lemma 3 *Let $\tau_p = \{T_1, T_2, \dots, T_n\}$ be a set of preemptable tasks, where $T_i = (C_i, S_i, D_i)$. C_i is the computation time, S_i the earliest start time and D_i the deadline of T_i . Let τ_p be sorted in non-decreasing order of start times and let τ_p be schedulable according to lemma 2. Then $\tau_p \cup T_0$ is schedulable if and only if:*

$$\forall j \left(\sum_{k=0}^j C_k \right) \leq D_i - S_0.$$

This lemma states that a task can be added to a set of previously guaranteed tasks given that there is enough processing time to guarantee the timely execution of this task and all the guaranteed tasks in the set. Since the tasks are preemptable, there might be several tasks with start times equal to the current time. In such cases we assume that the task being considered by the guarantee procedure precedes all other tasks with start times equal to the current time. The costs of preemption and running the dispatcher are ignored in this study, however, these costs can be considered as part of the task's computation time. Note that the incremental version of the guarantee procedure is valid only for task sets where the task that is being considered has start time equal to the present time and the only tasks that have start times in the future are periodic tasks that satisfy the schedulability condition of lemma 1.

III-C Stochastic-Learning Global Scheduling Strategy

The stochastic-learning strategy is based on a real-time extension of the stochastic-learning automata [4, 5, 8]. The stochastic-learning global scheduler at each processor of the distributed system keeps a variable \bar{L} that is proportional to the load on that processor. \bar{L} is defined as the fraction of the busy time to the total time in the current window of scheduling. This information is broadcast periodically as part of an update message which also contains the identification of the source of the message. The period of the update messages is a tunable system parameter. Using the update information received from other processors and the value of its own variable \bar{L} , the global scheduler at each processor can calculate the system's average load. It is then able to label each processor as *underloaded* or *overloaded*. This assignment

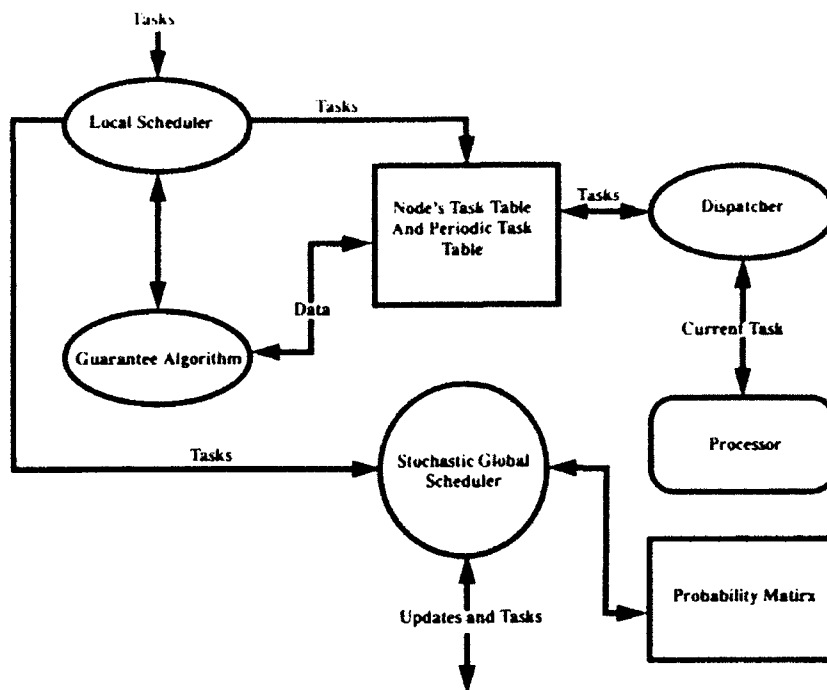


Figure 1: The operation of local and stochastic-learning global scheduling strategy for a processor of the distributed system.

of underloaded and overloaded to each processor constitutes the *state of the system* as observed by each global scheduler. Different processors of the system may observe different states of the system due to the different delays inherent in the communication medium.

The probability vectors for each possible state of the system are combined to form a probability matrix P , the rows of which correspond to the observed state of the system. An element of the probability matrix, P_{ij} at processor k , represents the probability that processor k will send an unguaranteed task to processor j should processor k find the system in state i . The diagonal elements are set to zero to prevent a processor from sending an unguaranteed task to itself. Furthermore, the probability of sending a task to a processor that is overloaded is set to zero. Initially unguaranteed tasks are sent to underloaded processors with equal probabilities.

The *learning* occurs as the result of processors guaranteeing or rejecting remote tasks. Let us assume that processor i 's estimate of the state of the system is S and that it has sent a task to processor j . If the task is not guaranteed, the action of sending an unguaranteed task to processor j , given processor i 's estimate of the state of the system, was an *incorrect* action. Therefore, the probability of i sending another task to j should be reduced when i finds the system in the same state, S . If the task was guaranteed by j , the probability of sending a task to processor j should be increased when i finds the system in state S . For both cases the row of P in processor i that is updated must correspond to the observed system state, S . Note that the source processor i sends its current observed state of the system, i.e. S , along with the task and processor j sends a message to inform processor i whether the task was guaranteed or rejected.

If the task was rejected then the elements of the effective row, i.e. row S , are updated by the following *penalty* function:

$$P_m(n+1) = P_m(n) + K_p \times P_m(n) \quad m \neq j$$

$$P_j(n+1) = P_j(n) - K_p \times \sum_{m \neq j} P_m(n)$$

If the task was guaranteed then the elements of the effective row, i.e. S , are updated by the following *reward* function:

$$P_m(n+1) = P_m(n) - K_r \times P_m(n) \quad m \neq j$$

$$P_j(n+1) = P_j(n) + K_r \times \sum_{m \neq j} P_m(n)$$

In the above equations n refers to the current value of the elements of the probability matrix and $n+1$ to the next instance of them. Note that the above computation is performed periodically as part of the global scheduler and that both K_p and K_r are tunable parameters.

Figure 1 is a pictorial representation of the local and global schedulers that reside at each processor of the distributed system. Each processor periodically, with period τ , checks and updates its local variable L and broadcasts it to all other processors in the system. Using this information, each processor determines the state of the system at this time. Once the local scheduler finds a task that it cannot guarantee then it sends that task to a remote processor using the row of the probability matrix that corresponds to the current observed state of the system. Upon the arrival of the remote task at the destination processor, the local scheduler of this processor attempts to guarantee the task. Rejected tasks are removed from the system and guaranteed tasks are entered into the NTT. The indication of the guarantee or rejection is sent back to the source processor and that processor updates the probability vector corresponding to its observed state of the system at the time of the initial transmission.

IV Simulation Model and Results

The simulation model, programmed in C, consists of a network of five homogeneous processors connected as shown in Figure 2. There are five independent sources for arrivals of aperiodic tasks. Each source is modeled by a Poisson distribution with averages $\lambda_1, \dots, \lambda_5$. The λ 's vary depending on the particular loads being modeled. Specific values of λ 's correspond to average loads which are presented with the simulation results later on in this section. When a task arrives at a processor from the external world, it is assigned a size according to a Poisson distribution. The average size of the tasks is another parameter that varies with the particular load being considered. If a task cannot be guaranteed by the processor at which it arrives from the external world, that task may be transferred to another processor of the network. The delivery of tasks and other messages, such as updates, take a time that depends on two factors, the queueing delay and the transmission delay. While the queueing delay depends on the behavior of each processor of the system the transmission delay is a physical property of the network and is a global variable chosen to reflect realistic situations in current networks. The queueing delay is determined by the simulation model.

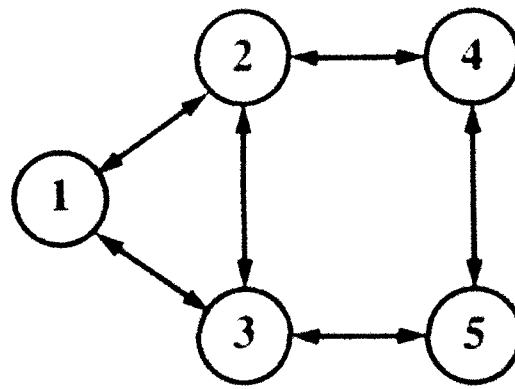


Figure 2: Network Topology

In a real-time system one of the most important parameters to consider is the *laxity* of tasks. The *laxity* of a task, T_i , is defined by: $LAX_i = D_i - C_i - R_i$, where D_i, C_i, R_i refer to the task's deadline, computation time, and ready time respectively. The laxity of all periodic tasks in the system is known in advance. The laxity of the aperiodic tasks is modeled as a Poisson distribution and is a system variable. In the simulations some of the processors of the distributed system are overloaded and others are not. In none of the cases however, is the system as a whole overloaded.

To evaluate the performance of the stochastic-learning scheduling strategy, two deterministic and three baseline global scheduling strategies are also implemented. The first deterministic strategy is a centralized strategy where one processor is responsible for making decisions about relocating tasks that cannot be guaranteed locally. The second deterministic strategy is a cooperative and distributed strategy in which a set of processors solicit and submit bids to acquire those tasks that cannot be guaranteed locally. The baseline algorithms consist of an optimal global scheduling strategy, a non-cooperative strategy in which tasks must be executed locally and a random global scheduling strategy. All global scheduling strategies use the local scheduler described in the previous section for local scheduling.

In the centralized global scheduling strategy a central processor is responsible for making global scheduling decisions. A task that cannot be guaranteed by the local scheduler is attached to a queue of locally unguaranteed tasks. The global scheduler at each non-central processor is responsible for dequeuing of these tasks. The central scheduler is a periodic task itself, but to decrease the number of late tasks it may also be invoked dynamically. The centralized global scheduling strategy is a three phase protocol. In the first phase of the protocol the global schedulers at non-central processors of the system inform the global scheduler at the central processor of the system of the locally unguaranteed tasks. In the second phase the central scheduler tries to find a suitable sink processor for each of the locally unguaranteed tasks. If a suitable sink processor is found its identity is sent to the source processor of that unguaranteed task. The third phase of the protocol starts when the source processor receives this message. The global scheduler on this processor labels the task remote and sends it to the designated sink processor. On the arrival of this remote task at the selected sink processor, if this task is guaranteed by the local scheduler, its timing information is inserted into the appropriate row of the NTT of the sink processor and the scheduler at the

central processor is informed. If this task is not guaranteed by the local scheduler, it is removed from the system and there is no need to inform the global scheduler at the central processor since the NTT of this processor remains unchanged. The centralized global scheduling strategy uses processor 2 as the central processor in the network of Figure 2. Note that this is the best possible location for the central processor as only one processor is not its immediate neighbor.

The cooperative strategy is an implementation of the bidding strategy introduced by Ramamritham and Stankovic [6, 9]. A new task that arrives into the system is considered first by the local scheduler. If that task is not guaranteed by the local scheduler, it is attached to a queue of unguaranteed tasks. The global scheduler is itself a periodic task but, it may also be invoked dynamically. The cooperative global scheduler also has three phases. During the first phase the global scheduler at the source processor generates and broadcasts a Request For Bids (RFB) message. In the second phase the global schedulers of other processors of the system receive the RFB message and may generate and send a bid to the source processor. In the third phase, the global scheduler at the source processor evaluates the bids and awards the task to the highest bidder. Once a remote task has been received by the sink processor, the local scheduler of the sink processor runs the guarantee procedure for this newly arrived remote task. If the task is guaranteed by the local scheduler the task is placed in the appropriate row of the NTT. If a remote task cannot be guaranteed by the local scheduler, it is rejected and removed from the system.

The first baseline strategy is an optimal strategy which assumes tasks can be processed at remote processors with zero communication and transmission delays. This strategy effectively replaces a homogeneous distributed system of N processors with a single processor which is N times faster than a processor of the distributed system. The second baseline strategy is a non-cooperative strategy. In this strategy tasks that cannot be guaranteed for local execution are rejected and removed from the system. The last baseline strategy implemented is a random global scheduling strategy which sends locally unguaranteed tasks to another processor selected at random from the distributed system. While the optimal global scheduling strategy defines the upper bound on the number of tasks that can be guaranteed in the system, the random and non-cooperative strategies define the lower bounds for the number of tasks that should be guaranteed by a global scheduling strategy.

A number of different metrics are measured in this study. The result reported, however, is only with respect to one metric, the percentage of the tasks that complete their execution prior to their assigned deadline. Since all periodic tasks are guaranteed to meet their timing requirements by the local scheduler, only the results for aperiodic tasks are considered. Each global scheduling strategy has a number of tunable parameters. All such parameters are optimized with respect to the normal operation of the network where none of the processors of the distributed system are highly overloaded. Once these parameters are optimized, their value is not changed.

Figure 3 shows the proportion of tasks guaranteed as a function of the average load of the system when the queuing delays are taken into account. Figure 4 shows the results in an unrealistic model in which queuing delays are ignored. As can be seen from these two figures, the queuing delay has a substantial effect on the guarantee ratio and the communication medium is a significant part of the network that cannot

be ignored in the design of distributed real-time scheduling algorithms. Furthermore, Figure 3 shows that the performance of the stochastic-learning strategy is superior to that of other algorithms for moderate to heavy loads. This phenomena is not observed in the unrealistic model of Figure 4 again, stressing the importance of communication medium and its real-time impacts.

In Figure 3 also note that the centralized scheme does not compare favorably with either the cooperative or the stochastic strategies. The stochastic scheduling strategy requires no messages to send or receive tasks. This reduces the delay in receipt of a task thereby increasing the probability of guaranteeing remote tasks. Both the centralized and the cooperative strategies require two messages before they can send a task to a remote processor. The cooperative scheme has, however, more up-to-date information and on the average achieves better results. Furthermore, the central processor of the system may also become congested, causing delay to request and update messages.

The stochastic-learning strategy lacks the up-to-date information of the cooperative strategy. For each locally unguaranteed task, however, the cooperative strategy sends two messages whereas the stochastic learning strategy sends only one. This increases the traffic of the network. Furthermore, the message sent by the stochastic scheduler is sent *after* the remote task has arrived at its destination. The cooperative scheduler requires two messages *before* it can send a task to a remote processor. The effect of these two factors is more pronounced for networks that have higher transmission delays or systems where the mean laxity of real-time tasks are smaller. Figure 5 shows the proportion of tasks guaranteed as a function of the average load of the system when the transmission delay is increased to 10 milliseconds per packet per hop. Each packet is 1000 bits long and the mean task size is 16000 bits. Comparing figures 5 and 3 shows that as transmission delay increases stochastic-learning scheduling strategy becomes more attractive compared with other strategies. Similar results can be deduced by comparing figures 6 and 3. Once again the stochastic-learning strategy is found superior to all other global scheduling strategies as the mean laxity of tasks is reduced from 1450 milliseconds to 500 milliseconds. Similar results are found when the sizes of tasks or the average load of the network are increased.

The results suggest some general conclusions about the effectiveness of stochastic and cooperative scheduling strategies. The stochastic scheduling strategy performs better than the cooperative strategy when the transmission delays are longer, the system is more heavily loaded, the laxity of tasks are smaller or the task sizes are larger. Thus, the stochastic scheduling is more suitable for *difficult* systems and it can be used in new applications where traditional deterministic schemes are not successful. On the other hand, the deterministic strategy is more suitable for *easy* systems that are not heavily loaded, tasks have more laxity, job sizes are smaller or transmission delays are not significant. It is important to note that there are situations where no global scheduling strategy performs well. For example if the size of tasks is too large, the communication delay will be too great regardless of the scheduling algorithm selected. It is also interesting to note that there are situations where any algorithm would be adequate. These include the trivial cases when none of the processors of the distributed system are overloaded and the case when transmission delays are exceedingly small.

V Conclusions and Future Research

This paper presents a novel dynamic scheduling strategy for real-time distributed systems. The problem of scheduling real-time tasks in a distributed system is decomposed into two related subproblems. First, a local scheduler attempts to guarantee the timely execution of real-time tasks by considering the current load on a single processor. Second, tasks that cannot be guaranteed locally are sent to another processor of the distributed system that has a high probability of executing remote tasks in time. The stochastic strategy *learns* the best possible action for each state of the network. Periodic messages are broadcast and collected to determine the state of the network. The stochastic-learning scheduling strategy is compared with several other global scheduling strategies. The simulation results demonstrate that stochastic-learning strategy is superior to deterministic strategies in many realistic situations and they also indicate that stochastic strategies extend the domain of real-time distributed controllers by successful scheduling tasks in *difficult* situations. In the future this research will consider heterogeneous networks and precedence and mutual exclusion relations among the real-time tasks.

Acknowledgments

The simulator was partially written by Y.S. Ramakrishna, Ching-yu Hung, Deb Agrawal and Huyuh Phu who also commented on various scheduling strategies. Their help is greatly appreciated.

References

- [1] Cheng, S-C., Stankovic, J. and Ramamritham, K., "Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey," *Hard Real-Time Systems*, Edited by John A Stankovic and Krithi Ramamritham, Computer Society Press, (pp. 150-173)
- [2] Dertouzos, M.L. and Mok, A.K.L., "Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks," *IEEE Transactions on Software Engineering*, vol. 15, no. 12 (December 1989), pp. 1497-1506.
- [3] Liu, C.L. and Layland J.W., "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment," *JACM*, vol. 20, no. 1 (January 1973), pp. 46-61
- [4] Mirchandaney, R., *Decentralized Task Scheduling Using a Network of Stochastic Learning Automata*, Master's thesis, Department of Electrical Engineering, Univ. of Massachusetts, Amherst, August 1984.
- [5] Narendra, K.S. and Thathachar, M.A.L., "Learning Automata-A Survey," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-4, no. 4 (July 1974), pp. 323-334.
- [6] Ramamritham, K. and Stankovic, J.A., "Dynamic Task Scheduling in Hard Real-Time Distributed Systems," *IEEE Software*, vol. 1, no. 3 (July 1984), pp. 65-75.
- [7] Sorenson, P.G., *A Methodology for Real-Time System Development*, Ph D Thesis, University of Toronto, June 1974.
- [8] Stankovic, J.A., "Stability and Distributed Scheduling Algorithms," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 10 (October 1985), pp. 1141-1152.
- [9] Stankovic, J.A. Ramamritham, K. and Cheng, S-C., "Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems," *IEEE Transactions on Computers*, vol. 34, no. 12 (December 1985), pp. 1130-1143.

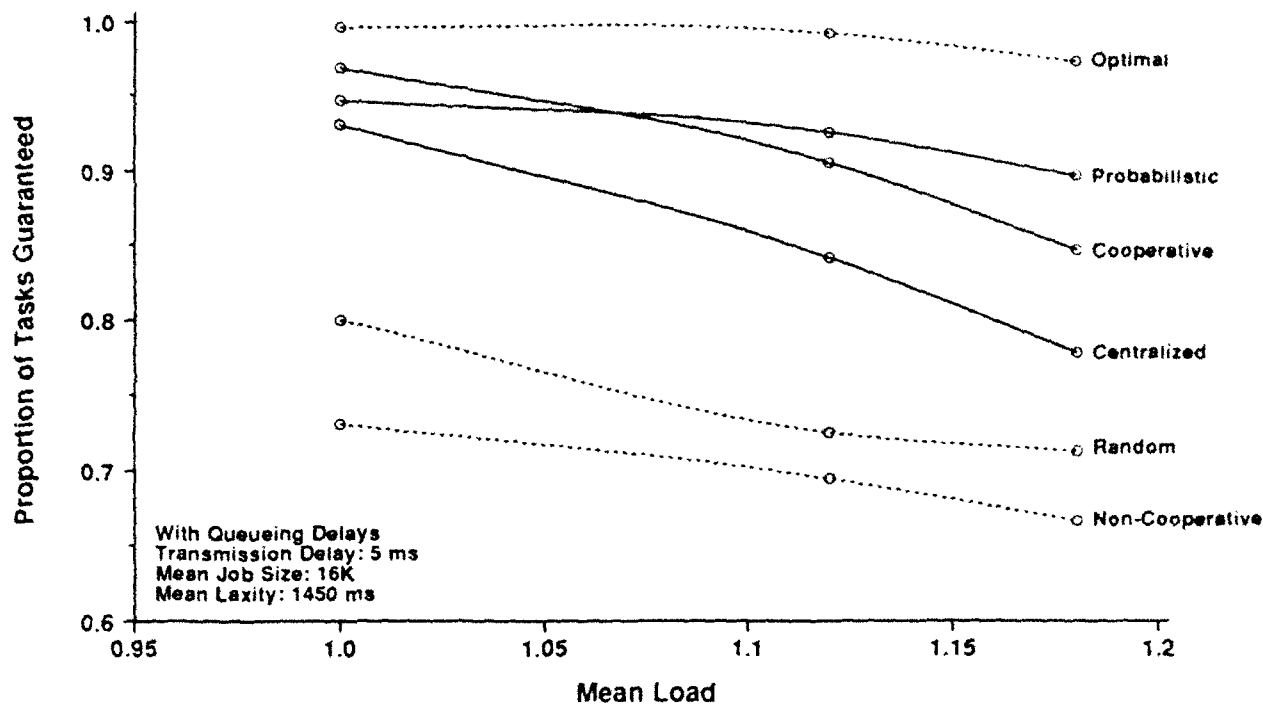


Figure 3: Proportion of tasks guaranteed as a function of mean load on each node of the distributed system. In this figure the queueing due to the congestion of the links is modeled. Note that the optimal algorithm does not incur any form of communication delays and hence, does not represent a realistic network.

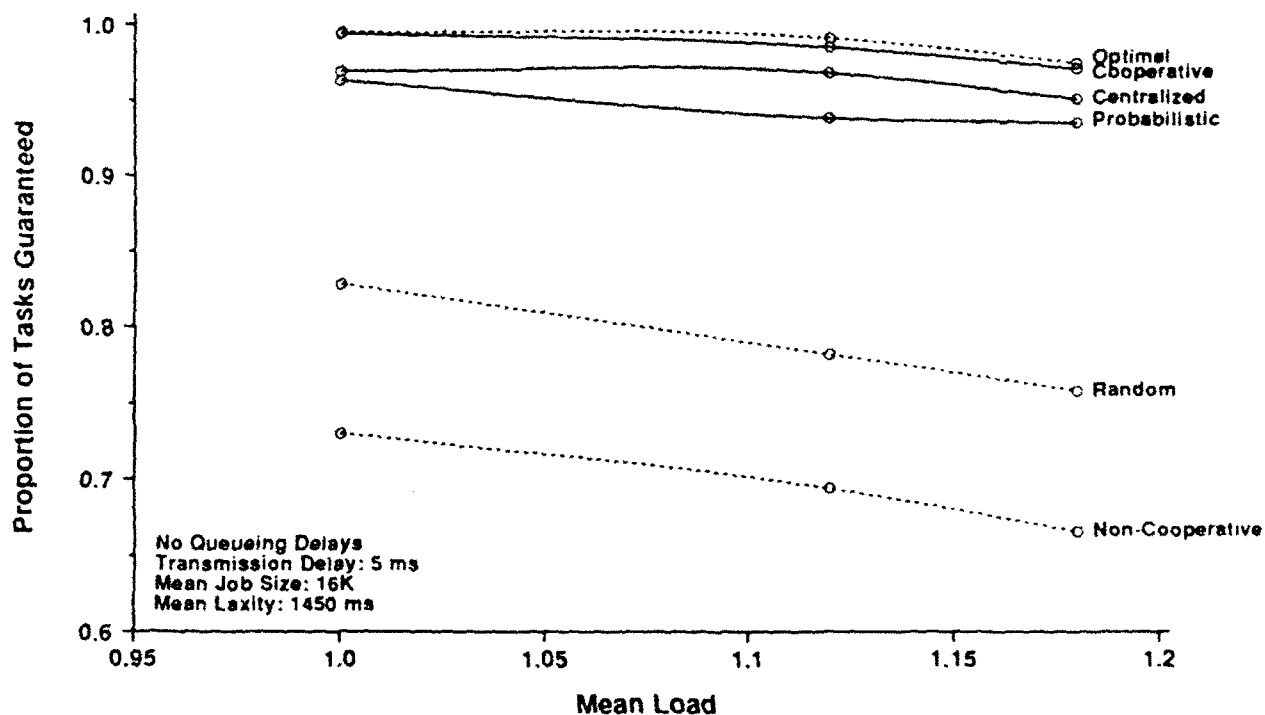


Figure 4: Proportion of tasks guaranteed as a function of mean load on each node of the distributed system. In this figure the queueing is ignored.

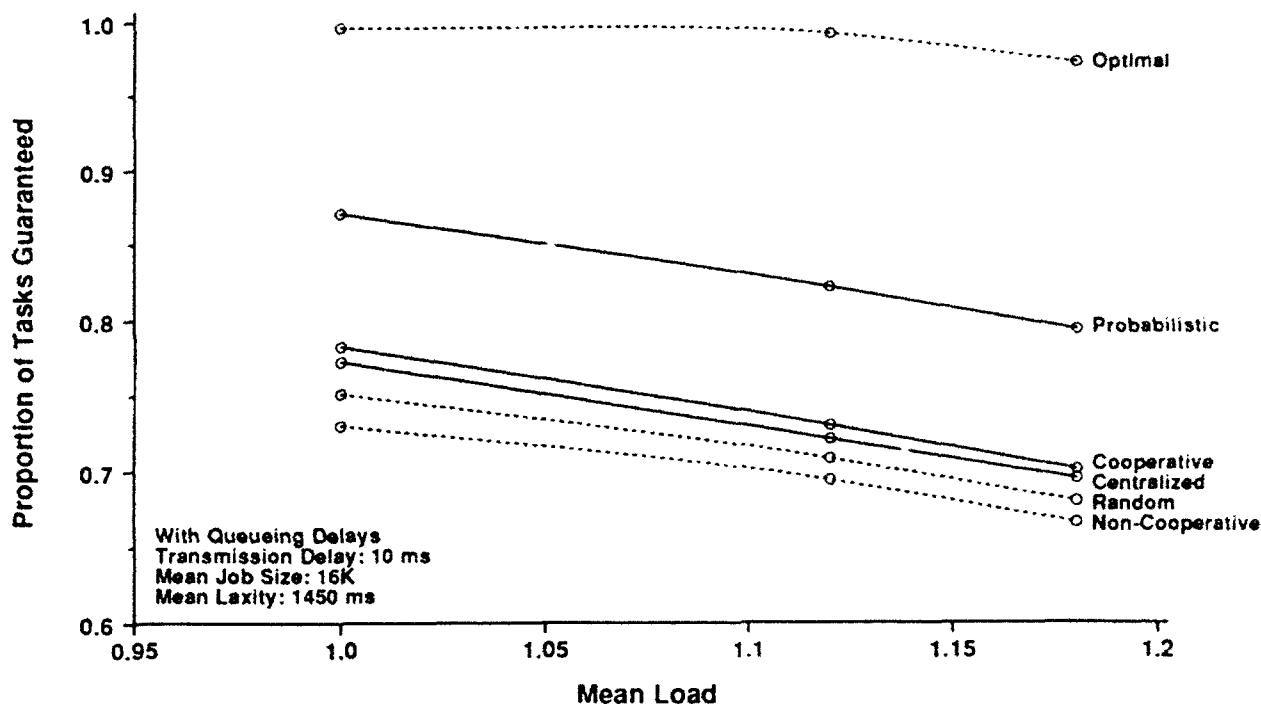


Figure 5: Proportion of tasks guaranteed as a function of mean load on each node of the distributed system. In this figure the transmission delay is increased to 10 milliseconds per packet per hop.

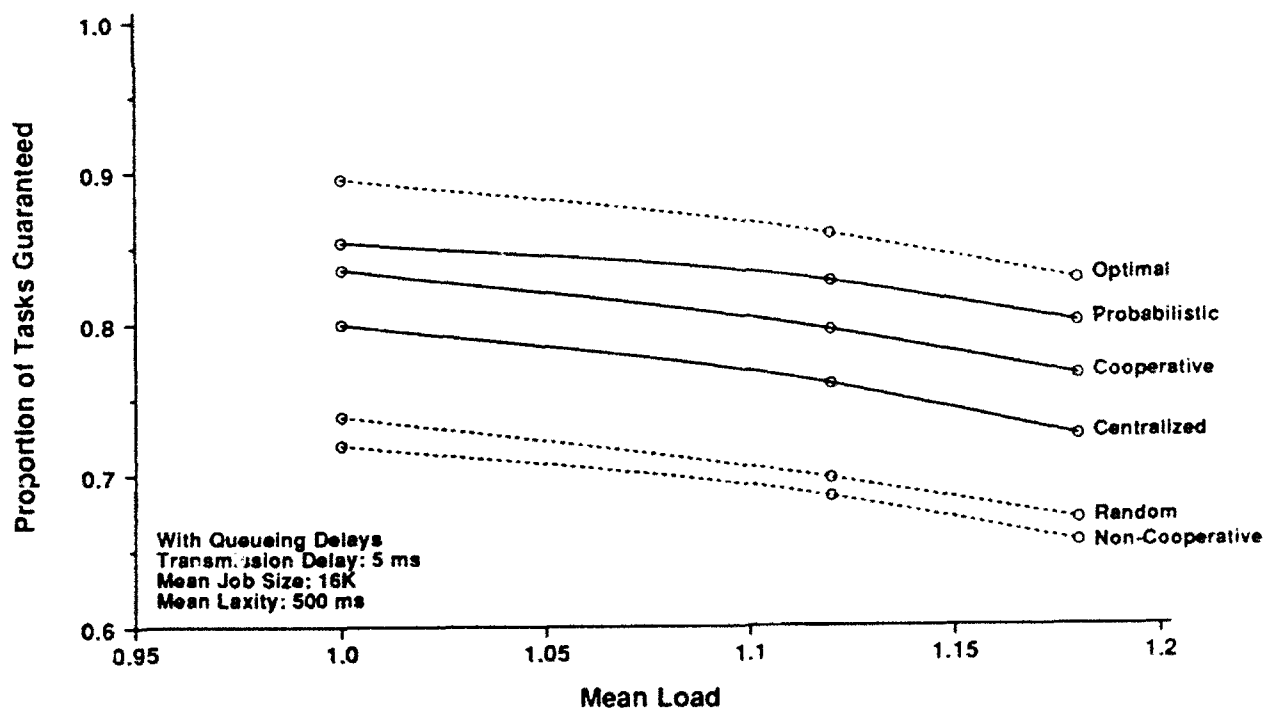


Figure 6: Proportion of tasks guaranteed as a function of mean load on each node of the distributed system. In this figure the mean laxity of tasks is reduced to 500 milliseconds.

SYSTEM ENGINEERING I

Massively Interconnected Models for a Beam Former

Chin-Hwa Lee
Naval Postgraduate School
Code EC/LE
Monterey, CA 93943
chlee@ece.nps.navy.mil

1. MIM: Conceptual decomposition

When a mathematical model is translated to a detailed structural model for implementation, often the communication consideration outweighs the computational considerations. Such a structural model is referred to as the *Massively Interconnected Models* (MIM). Numerous examples exist in the areas of parallel computation, distributed computer communication networks, and artificial neural networks. The beam forming problem in acoustic array processing is one kind of Massively Interconnected Model, which challenges the designers who hope to exploit parallel processing technology of the future.

Beam Forming is a spatial filtering problem. Signals are coming in from multiple acoustic sensors numbered in thousands that are placed over a spatially limited area (2D). These signals are processed and enhanced coherently so that directivity of the arrival beam can be discriminated in both the azimuth and elevation directions. These signals will be transformed and manipulated in a mathematical model that requires MIM for implementation. The beam forming problem is selected here to demonstrate a procedure necessary to describe typical MIM problems. It is used also to develop characterization that can differentiate among available architectures which can fit the MIM implementation.

In this paper, the beam forming problem is first considered in a mathematical model. At this stage mathematical abstract formulae and notations are used to describe the algorithm. Later, to implement this algorithm a practical architecture was selected to accommodate the processing. The architectures may be VLSI chips, bit-slice micro-programmable processor, multiple processor systems using DSP chips or Floating Point Unit hardware (FPU), SIMD machines such as CM-2, or MIMD machine such as Hypercube or iWarp. Modeling using MIM is an intermediate step in which alternative partition of the algorithm is studied. Vital issues in this step is the occurring resource allocation and scheduling when MIM modules are fitted to the architectures.

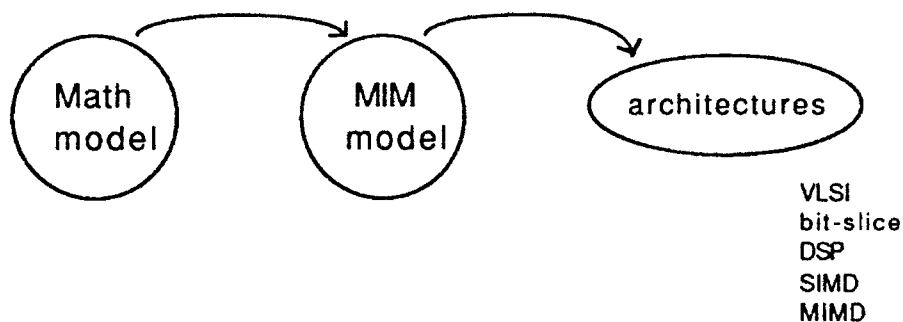


Figure 1. Design procedures of a beam former.

Therefore, MIM modules are not considered as direct hardware architectural components but as a conceptual partition of the algorithm. This decomposition into MIM modules results in resource allocation and scheduling problems.

Since the objective is to implement a signal processing algorithm (Beam forming) in hardware following a most economical way that includes the whole life cycle cost (hardware/software design cost and maintainability costs), it is necessary to develop *measures* that can help to answer the following questions:

1. What are the capacity requirements in the DSP algorithm?
2. Among all the architectures, such as VLSI, bit-slice, DSP chips, SIMD machines, and MIMD machines, which architecture can fit the beam former best? This is a complicated issue. The MIM module decomposition is used to resolve this question.
3. How do you decompose the algorithm into MIM modules? The decomposition depends very much on the architecture under consideration. How are these considerations related?

2. Beam Forming Algorithm

Let's assume there is a source in the far field radiating a planar wave at an angle of θ with respect to the sensor array shown in Figure 2. The signal received at sensor 1, $x_1(t)$ is a delayed version of the signal $x_2(t)$, $x_3(t)$, ..., $x_M(t)$ of the other sensors. Let the delay in time be Δ

$$\Delta = d \cos \theta / \lambda \quad (1)$$

where λ is the wavelength of the source. d is the distance between

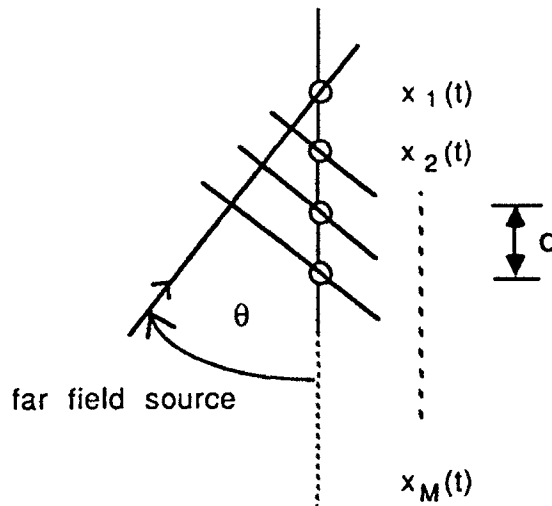


Figure 2. Linear uniformly spaced sensor array.

sensors. Assume that we are dealing with an evenly spaced linear array of sensors. The distance between sensor is d .

$$\begin{aligned}
 x_1(t) &= x_0(t-\Delta) \\
 x_2(t) &= x_1(t-\Delta) = x_0(t-2\Delta) \\
 &\vdots \\
 x_M(t) &= x_{M-1}(t-\Delta) \dots = x_0(t-M\Delta)
 \end{aligned} \tag{2}$$

If we consider the Fourier Transform of the sensor signals,

$$\begin{aligned}
 X_1(f) &= X_0(f) e^{-j2\pi f\Delta} \\
 X_2(f) &= X_0(f) e^{-j2\pi f2\Delta} \\
 &\vdots \\
 X_M(f) &= X_0(f) e^{-j2\pi fM\Delta}
 \end{aligned}$$

or

$$X_n(f) = X_0(f) e^{-j2\pi f n\Delta} \quad (3)$$

Here is the broad-band acoustic beam forming procedure that uses basically a delay-sum approach. There are two possible approaches for broadband beam forming procedure; time domain approach and frequency domain approach. Due to easier computation requirements and applicability to *optimal* and *adaptive* beam forming, the frequency approach is considered here. The signal flow diagram for a beam former is shown in Figure 3. For each frequency a set of $W_n(f)$ is used in the algorithm.

$$Y(f) = \sum_{n=1}^M a_n X_n(f) \cdot W_n^*(f) \quad (4)$$

Select,

$$W_n(f) = e^{-j2\pi f n\Delta} \quad (5)$$

Equation (4) becomes.

$$\begin{aligned} Y(f) &= \sum_{n=1}^M a_n \cdot [X_0(f) e^{-j2\pi f n\Delta}] \cdot W_n^*(f) \\ &= \sum_{n=1}^M a_n \cdot X_0(f) e^{-j2\pi f n\Delta} \cdot e^{j2\pi f n\Delta} \\ &= \sum_{n=1}^M a_n X_0(f) = X_0(f) \sum_{n=1}^M a_n \end{aligned}$$

It is obvious that this special selection of $W_n(f)$ resulted in the coherent summation of the beam former output, $Y(f)$. The $W_n(f)$ is called the *steering vector* that is dependent on the angle of arrival θ in equations (5) and (1). In order to hear the beam from different directions the whole operation in Figure 3 are repeated for each direction angle. The steering vector in reality also depends on the source frequency as shown in equation (5). Therefore, even though the direction of the beam is fixed,

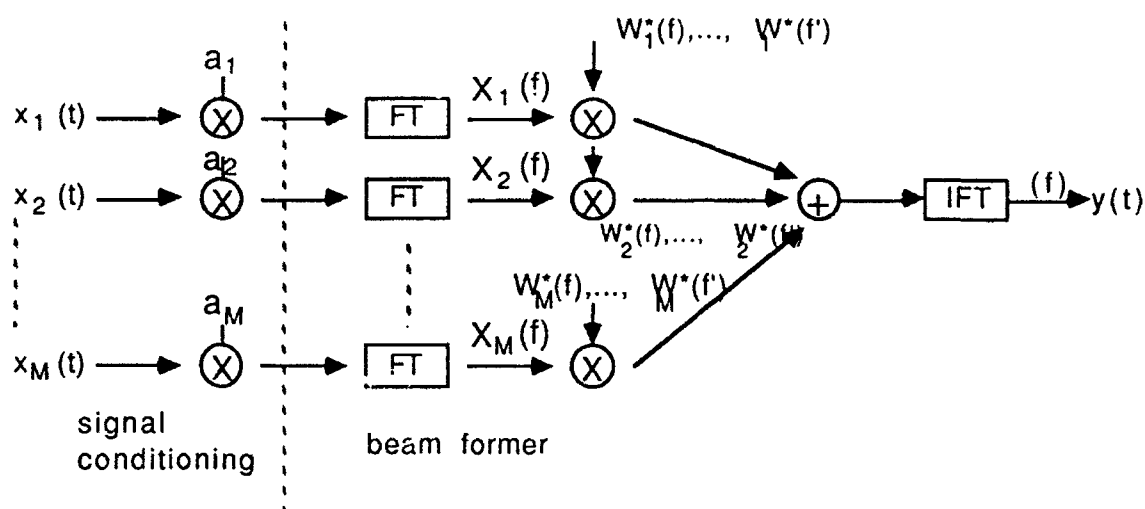


Figure 3. Frequency domain approach for beam former.

different sets of $W_n^*(f)$ have to be provided for a different temporal frequency in the broad band spectrum. If audio output is required, the spectrum signal $Y(f)$ will be inversely transformed into the time domain.

3. To Answer the Obvious

Many people know the answer to the question of whether the beam former is a computation intensity operation. Here a set of *measures* are developed to answer this obvious question. The computation requirements of a beam former is determined from the mathematical model developed in the previous section. The model shown in Figure 3 encompasses the equations and English description of the previous section. It is mathematical, but not vigorous and rigid from the point of view of a computer language syntax. In order to show it as a computation intensive operation we use the following measures.

M1: Computation Bandwidth (BW) requirement: A real-time beam former receives acoustic signals, processes the data, and provides beam information $y(t)$ for a specific frequency to subsequent systems. Due to hard real-time delays required for processing it is necessary to consider processing capacity in unit time. Frequency of operations per unit time is characterized in terms of the computational BW requirement (Megaflops/sec).

M2: Communication Bandwidth (BW) requirement: The beam former typically processes data from sensors and steering vectors from database. The hard real-time delay has to be satisfied. These are specified as I/O bandwidth (BW) requirements (bytes/sec). The beam former algorithm requires certain capabilities, and whether an architecture can fulfill that requirement is an important question.

M3: Memory Bandwidth Requirement: Very often in order to speed up operation some of the coefficients can be precalculated and stored into the memory. Sometimes, the memory reads and writes are so frequent and intensive, the total run time slows down. In real-time applications it is also imperative to characterize this impact in terms of memory bandwidth requirement.

4. Beam Former? Be Specific!

A beam former (BF) problem is discussed here. The requirement of this beam former depends very much on the size of the array and the number of beams involved. It is essential to be specific to analyze a beam former problem. A Passive Sonar Example Draft (version 0.02) created for the Design and Synthesis Technology Project in Naval Surface Warfare Center is used here as a guideline. This example allows fixed beam forming, steerable beam forming, and acoustic channel output. Environmental and seasonal data can also influence the operation of the BF. For the study here to demonstrate the idea of MIM decomposition, this draft system has been simplified as follows.

Very Simplified Beam Former:

1. Only fixed beam forming is included.
2. A 1-D linear uniformly spaced array is considered.
3. Total sensors is 100.
4. Frequency domain approach is adopted.
5. Azimuthal coverage of 360° with 3° resolution.
6. Frequency coverage is from 0-256 Hz.

The coefficient multiplication with a_n in equation (4) is sensor dependent. This multiplication is part of the signal conditioning operation. It is done in analog electronics. Therefore, it is not considered as part of the beam former. There are two basic kinds of operations in a beam former. The first operation is the Discrete Fourier transform using Fast Fourier Transform (FFT) algorithms. To cover a spectrum to 256 Hz, Nyquist rate requires a sampling frequency of

$f_s = 1/T = 512$ Hz. Let's assume that N-point FFT is involved in this system. The total rate for doing FFT is

$$R_{FFT} = \frac{N_e \cdot 1/2 \cdot N \log_2 N}{NT} = 1/2 \cdot N_e \log_2 N \cdot f_s$$

where: N_e is the number of sensor elements, i.e. $N_e = 100$.

$$N = 512 \quad \text{and} \quad T = 1/512 \text{ sec}$$

Because real data is transformed, the Hermitian property spares us from calculating negative frequency components. Therefore, the total calculation is $1/2 \cdot N \cdot \log_2 N$ within $N \cdot T$ sec for all sensor signal. For our very simplified BF,

$$R_{FFT} = 1/2 (100) \cdot 9 \cdot 512 = 230.4 \text{ KFLOPS}$$

The second operation is the dot product calculation shown in equation (4). There are N_e number of multiply-add operations for each beam direction. Since the steering vector depends also on the frequency, with N frequency the rate for vector product is

$$R_{DP} = \frac{N_b N_e N}{NT} = N_b \cdot N_e \cdot f_o$$

where N_b is the number of beams, i.e. $N_b = 120$. For the very simplified BF.

$$R_{DP} = 120 \cdot 100 \cdot 512 = 6.144 \text{ MFLOPS}$$

The total computational bandwidth of the BF is

$$\text{CompBW} = R_{FFT} + R_{DP} = 6.385 \text{ MFLOPS}$$

The beam former is one part of the total system. Its effectiveness depends on the input/output it can provide. The input bandwidth requirement comes from accepting the signals from the conditioner.

$$\text{I/P BW} = N_e \cdot f_s \cdot 2 \text{ bytes/sample}$$

$$= 100 \cdot 512 \cdot 2 = 102.4 \text{ Kbytes/sec}$$

Where, each sample from A/D convertor is a 12-bit real value which takes 2 bytes. The output is fed to the *tracker* process of the simplified sonar system. Each beam data has 4 bytes per sample.

$$\text{O/P BW} = N_b \cdot f_s \cdot 4 \text{ bytes/sample}$$

$$= 120 \cdot 512 \cdot 4 = 245.76 \text{ Kbytes/sec}$$

While the dot product is in operation, it is necessary to read out the steering vectors for equation (5) in real-time. The memory bandwidth requirement is,

$$\text{MEMBW} = N_b \cdot N_e \cdot f_s \cdot 4 \text{ bytes/output}$$

$$= 120 \cdot 100 \cdot 512 \cdot 4$$

$$= 24.576 \text{ Mbytes/sec}$$

In summary, the mathematical model can be represented as a VHDL entity with its capacity requirement as shown in Figure 4.

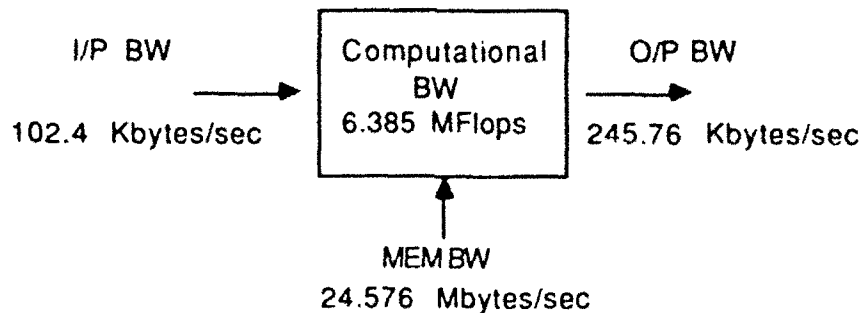


Figure 4. Mathematical model of beam former with capacity requirement.

5. Peak rate and sustainable rate

As a first look of the capacity requirement shown in Figure 4, it does not seem to be stringent. Keep in mind that the very simplified beam former is a trivial example. Realistic beam former has much larger capacity requirements than these. Even for such a small task the bandwidth described so far is the *sustainable* bandwidth not the *peak* bandwidth. Many commercially available processors or DSP chips claim a capacity in this order of magnitude. But, their claims are universally *peak* bandwidth. It means under the most ideal situations without delay of data and delay of instructions the processor can achieve, for example, 10 MFLOPS capacity. The overall sustainable processing rate very much depends on the communication of data or instructions in the system, which is much smaller than the peak rate. Consequently, this is a MIM type of situation where the communication consideration outweighs the computational considerations in the system. Exactly how communication affects the computation depends on the type of algorithms involved. Some commercial systems can achieve a very high CPU processing rate (FLOPS). But, sustainable rate for other types of jobs is very poor. As far as beam forming is concerned sustainable rate of a potential candidate implementation or architecture is of primary interest.

7. Measures characterizing architectures

Before we attempt to address the question of how to partition algorithm into MIM modules, it is necessary to characterize the MIM modules to see whether it can be accommodated in a particular architecture. In addition to the bandwidth requirement developed previously, it is also essential to consider the following measure.

M4: FLOPS-I/O ratio (α): This is a measure to characterize the proportion of computation done versus communication (I/O) required in the partition. It can be used to describe the MIM module requirement. It can also be used to characterize the architecture element. If the peak FLOPS-I/O ratio (α) of an architecture element is less than the MIM module requirement, it is possible to fit the MIM module into the element. If the peak FLOPS-I/O ratio (α) of an element is greater than the MIM requirement, problem exists to use this architecture element.

M5: Latency -FLOPS product (β): This is a measure to compare different decompositions of MIM modules for architectural elements. Generally, an element can be tuned to achieve a high FLOPS rate, but the latency associated with the data stream is also increased. Consequently,

it is necessary to compare the latency-FLOPS product among different architecture elements. For a particular MIM module partition it is also necessary to calculate the latency-FLOPS product. Only elements with less latency-FLOPS product can accommodate the MIM module of larger latency-FLOPS product.

MIM modules with small FLOPS-I/O ratio and latency-FLOPS product are generally referred to as *fine grain* tasks. On the other hand, architecture elements usually have limited achievable FLOPS-I/O ratio and latency-FLOPS product. Fine grain MIM modules can only be accommodated in fine grain architecture elements. Essentially, the FLOPS-I/O ratio and the latency-FLOPS product are measures to characterize computational activities relative to communication activities. With these measures it will be easier to analyze the results of different mapping approaches.

8. MIM decompositions

Let's assume that a real beam former is probably a hundred times bigger than the very simplified example considered here. Realistically it is not possible to use a high performance processor to accommodate the capacity requirement shown in Figure 4. Either specialized hardware, multiple processor, or parallel processor systems have to be used to accommodate the problem. The issue is about how to partition the BF algorithm and mapping them to a specific architecture.

Up to now the possible known implementations of a beam former can be summarized in Table 1. Most beam formers are implemented in bit-slice microprocessors with FFT and FPU chips [1]. Communication is done point-to-point through high speed short run-length buses. The architecture is arranged by the designer. There is some micro programming involved. It is small and just for the purpose of controlling the beam former. With all the FFT chips available such as TRW2310, HDSP66110, and UT69532, multiple-bus structure using this kind of chips has been developed for beam forming. Real time operation of this kind of system has been demonstrated. Beam forming was done on a Single Instruction Multiple Data (SIMD) parallel system such as CM-2, [2]. Even for non real-time prototypes, the programming work is not easy. Other kinds of parallel systems such as Multiple Instruction Multiple Data (MIMD) were used to prototype a beam former, [4]. EMSP is one of the Navy system tried over the years. Programming and communication scheduling is an important issue in this kind of system. The overall objective is to develop a low cost system for both hardware and software costs over the life cycle of the system. More work needs to be done to demonstrate these new technology implementations.

For an algorithm the MIM decomposition depends on the mathematical

model and the available architectural elements. In Figure 1 to do a good MIM decomposition it is necessary to consider both the left hand side algorithm and the right hand side architectural element. For the bit-slice processor implementation, you probably will decompose the mathematical model into a MIM module more along the line of the signal flow graph shown in Figure 3. Basically the beam former will have the following architectural elements.

- (a) FFT chips for DFT algorithm.
- (b) FPU chip for floating point arithmetic.
- (c) Point-to-point High Speed bus for communication.
- (d) Bit-slice microprocessor to implement control sequences.

If you are considering a mesh connected iWarps for beam former implementation, a different approach for MIM decomposition may be used. An iWarp processor is more capable than doing a simple FFT job. The MIM module can be a *combination* of FFT's, floating point multiply and addition together. The crucial issue is on what will be the grain size of the MIM module, and can it be fitted into an iWarp architectural element. The approach is to calculate the measures discussed previously for both the MIM modules and the iWarp processor. Then, fitting is based on comparing the measures of both the MIM modules and the iWarp processor.

9. MIM modules for bit-slice Implementation

Consider the FFT module for a bit-slice implementation shown in Figure 3. The total computational bandwidth required is

$$\begin{aligned} \text{CompBW} &= \frac{1/2 \cdot N \log_2 N}{NT} = 1/2 \cdot \log_2 N f_s \\ &= 2,304 \text{ FLOPs} \end{aligned}$$

This requirement is equivalent to do a 512 point FFT transform in one second. In reality an FFT module in a board can perform a 512 point FFT in 10 msec continuously. It means that if only computation BW is concerned, a FFT board module can accommodate 100 MIM modules.

In reality, it is very important to consider the communication bandwidth requirement as well. For the FFT MIM module in Figure 3. Input and output bandwidth requirements are as follows:

$$\begin{aligned} \text{I/P-BW} &= 512 \cdot 2 \text{ bytes/sec} \\ &= 1,024 \text{ bytes/sec} \end{aligned}$$

$$\begin{aligned} \text{O/P-BW} &= 512 \cdot 4 \text{ bytes/sec} \\ &= 2,048 \text{ bytes/sec} \end{aligned}$$

$$\text{CommBW} = \text{I/P-BW} + \text{O/P-BW} = 3,072 \text{ bytes/sec}$$

For example an array processor board such as PL2500 from Eighteen Eight Laboratory, only the peak bandwidth on an AT bus was known as 3.3 Mbytes/sec. Considering sustainable rate and all of the communication overhead, 1% of the peak rate was chosen as a nominal rate, 30Kbytes/sec. Consequently, a FFT hardware board can only accommodate 10 FFT MIM modules. This situation is described in Figure 5. In reality, both computational bandwidth and communication bandwidth are considered. The FFT board can only accommodate 10 channels of FFT MIM modules so far.

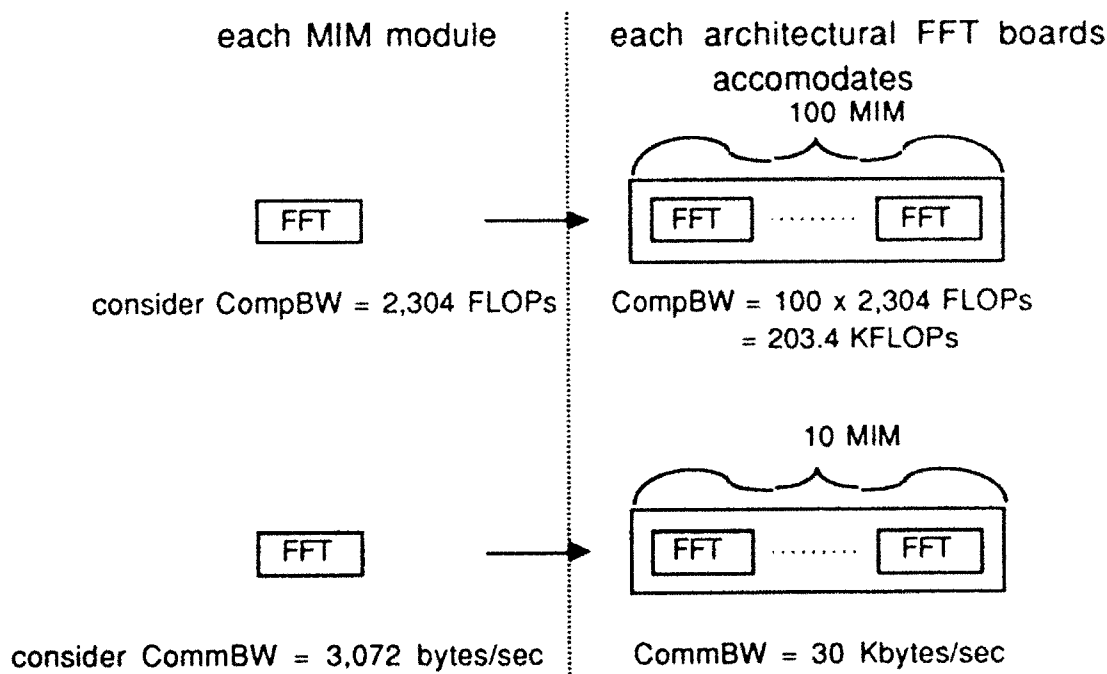


Figure 5. Mapping of MIM to architectural elements with partial consideration of measures.

10. Conclusions

The algorithm and characteristics of a beam former was introduced

and described. Due to the nature of a beam former, communication considerations outweigh the computational considerations in this problem. That is a typical Massively Interconnected Modeling (MIM) type of problem. A set of capacity measures in terms of bandwidth are developed here. These measures are used in the mapping process from mathematical algorithms to MIM modules and from MIM modules to architectural elements. Due to available space for discussion only the bit-slice approach for a beam former is presented in a simple analysis using the capacity measures. It shows that communication considerations are the main factor in all fitting/mapping problems for a beam former.

References

1. Signal Processing Handbook, edited by Chen, "Digital Time Delay Beamforming," Roger Pridham, Dekker, 1988.
2. B.D. Van Veen and K.M. Buckley, "Beamforming: A Versatile Approach to Signal Filtering," IEEE ASSP Magazine, April 1988.
3. W. Anderson, G. Lamb, W. Smith, "Acoustic Signal Processing on a Connection Machine," NRL Report 9285, Nov. 28, 1990.
4. S. Borkar, et al., "Supporting Systolic and Memory Communication in iWarp," proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, WA, May 28, 1990.

BF Implementation	Existence	communication programmability/ difficulty	Computational programmability/ difficulty	Hardware/ software Engineering cost	Hardware/ software Maintainability cost
VLSI	Not realistic	.	.	prohibitive	.
bit-slice with FFT and FPU	yes	fixed/low	fixed	High	Med.
Multiprocessor, DSP chips (DSP3, i860)	prototype non-realtime	prog/high	prog/?	Med.	High
SIMD (CM-2) [3]	prototype non-realtime	prog/serious	prog/?	Med.	High
MIMD (iwarp [4], EMSP)	prototype	prog/high	prog/?	Med.	High

Table 1. Known beam former implementations.

Analyzing Concurrent and Fault-Tolerant Software using Stochastic Reward Nets

Gianfranco Ciardo

Jogesh K. Muppala

Software Productivity Consortium

Software Productivity Consortium

Herndon, VA 22070

Herndon, VA 22070

Kishor S. Trivedi *

Dept. of Electrical Engineering

Duke University

Durham, NC 27706

*This work was supported in part by the National Science Foundation under Grant CCR-9108114 and by the Naval Surface Weapons Center under the ONR Grant N00014-91-J-4162.

Author for Correspondence:

Dr. Gianfranco Ciardo

Software Productivity Consortium

SPC Building

2214, Rock Hill Road

Herndon, VA - 22070

Tel.: (703) 742 - 7129

Fax.: (703) 742 - 7200

E-Mail: ciardo@software.org

Abstract

We present two software applications and develop models for them. The first application considers a producer-consumer tasking system with an intermediate buffer task and studies how the performance is affected by different selection policies when multiple tasks are ready to synchronize. The second application studies the reliability of a fault-tolerant software system using the recovery block scheme. The model is incrementally augmented by considering clustered failures or the effective arrival rate of inputs to the system.

We use stochastic reward nets, a variant of stochastic Petri nets, to model the two software applications. In both models, each quantity to be computed is defined in terms of either the expected value of a reward rate in steady-state or at a given time θ , or as the expected value of the accumulated reward until absorption or until a given time θ . This allows extreme flexibility while maintaining a rigorous formalization of these quantities.

List of Symbols

π	greek pi (lower case)
Π	greek pi (upper case)
λ	greek lambda
μ	greek mu
ν	greek nu
δ	greek delta
θ	greek theta
τ	greek tau
ρ	greek rho
σ	greek sigma
α	greek alpha
β	greek beta
ξ	greek xi
w	w (lower case)
\emptyset	empty set, similar to a zero with a slash through it
\mathbb{R}	strange R, for real numbers
\mathbb{N}	strange N, for natural numbers
$\#$	pound sign
∞	infinity sign
\forall	forall
\in	membership in a set
D^-, D^+, D°	D sup minus, D sup plus, D sup small circle

1 Introduction

Many applications demand high performance and reliability/availability from computer systems. Higher levels of integration and newer techniques in VLSI design have made hardware with high performance and reliability, relatively inexpensive. Software, on the other hand, is becoming a major component in the overall cost of these systems [27]. Often, though, the software poses performance and reliability bottlenecks which should be discovered and eliminated. Improvements in software assessment methods for the design phase of the software life cycle are required to minimize costly redesigns and changes due to unanticipated performance or reliability problems.

Markov models have been used for software performance assessment [10], software reliability assessment [17], and for analyzing software fault-tolerance [9, 14, 21]. Markov models have been quite popular in hardware performance models and hardware reliability models as well. Reasons for the popularity of Markov models include the ability to capture various dependencies, the equal ease with which steady-state, transient and cumulative transient measures can be computed and the extension to Markov reward models useful in performability analysis. The main drawbacks of Markov models include the size of the state space and the assumption of exponentially distributed sojourn times. It is possible to remove the assumption of exponential sojourn time distributions by using phase-type expansions of non-exponential distributions [13, 29]. This method converts a non-Markovian problem into a Markovian one with an even larger state space.

Stochastic Petri nets (SPNs) can be used to specify the problem in a concise fashion and the underlying Markov chain can then be generated automatically. Algorithms for storing and efficiently solving relatively large Markov chains have emerged and have been implemented in several packages [6, 8, 23]. Our version of SPNs, called stochastic reward nets (SRNs), not only allows the compact specification of large Markov models but also permits the concise specification of reward structure at the net level. In this way, automatic generation of large Markov reward models is facilitated. Steady-state, transient, and cumulative transient measures of the resulting Markov reward models can be computed [8]. We illustrate our approach with

two examples: software performance assessment of a producer-consumer system and reliability assessment of the recovery block, a software fault-tolerance scheme.

As we will show, detailed behavior of the system can be described concisely and the effects of various design decisions can be predicted easily. We note that SRNs are also suitable for hardware performance, reliability, and performability analysis, hence they can be used for combined hardware-software analysis. Some aspects of system hardware are indeed represented in the models described in this paper.

Several papers are relevant to our study. Performance modeling of concurrent software has been carried out using Markov chains [11, 12], series parallel graphs [16], queueing networks [28], stochastic rendezvous networks [31, 30], and SPNs [18, 5, 26]. A recent study by Leu *et al.* uses SPNs to model fault-tolerant aspects of software [15].

Section 2 gives a brief review of the SRN concepts; it also contains an explanation for the symbols used in the paper. In Section 3 we present the analysis of a producer-consumer tasking system and in Section 4 we present the analysis of the recovery block scheme. Conclusions are presented in Section 5.

2 Stochastic Reward Nets

There are several definitions for Petri nets [19, 20] and even more for stochastic Petri nets. Our SRN formalism allows only exponentially distributed or constant zero times, so its underlying stochastic process is independent semi-Markov with either exponentially distributed or constant zero holding times. We assume that the semi-Markov process is regular, that is, the number of transition firings in a finite interval of time is finite with probability one. Such a process can then be transformed into a continuous-time Markov chain as it is done for the generalized stochastic Petri net (GSPN) formalism [1].

The SRNs differ from the GSPNs in several key aspects. From a structural point of view, both formalisms are equivalent to Turing machines. But the SRNs provide enabling functions, marking-dependent arc cardinalities, a more general approach to the specification of priorities, and the ability to decide in a marking-dependent fashion whether the firing time of a transition

is exponentially distributed or null, often resulting in more compact nets. Perhaps more important, though, are the differences from a stochastic modeling point of view. The SRN formalism considers the measure specification as an integral part of the model. Underlying an SRN is an independent semi-Markov reward process with reward rates associated to the states and reward impulses associated to the transitions between states. Our definition of SRN explicitly includes parameters (inputs) and the specification of multiple measures (outputs). A SRN with m inputs and n outputs defines a function from \mathbb{R}^m to \mathbb{R}^n .

We define a non-parametric SRN as an 11-tuple $A = \{P, T, D^-, D^+, D^\circ, e, >, \mu_0, \lambda, w, M\}$, where:

- $P = \{p_1, \dots, p_{|P|}\}$ is a finite set of places. Each place contains a non-negative number of tokens. The multiset describing the number of tokens in each place is called a marking. The notation $\#(p, \mu)$ is used to indicate the number of tokens in place p in marking μ . If the marking is clear from the context, the notation $\#(p)$ is used.
- $T = \{t_1, \dots, t_{|T|}\}$ is a finite set of transitions ($P \cap T = \emptyset$).
- $\forall p \in P, \forall t \in T, D_{p,t}^- : \mathbb{N}^{|P|} \rightarrow \mathbb{N}, D_{p,t}^+ : \mathbb{N}^{|P|} \rightarrow \mathbb{N}$, and $D_{p,t}^\circ : \mathbb{N}^{|P|} \rightarrow \mathbb{N}$ are the marking-dependent multiplicities of the input arc from p to t , the output arc from t to p , and the inhibitor arc from p to t , respectively. If an arc multiplicity evaluates to zero in a marking, the arc is ignored (does not have any effect) in that marking.

We say that a transition $t \in T$ is arc-enabled in marking μ iff

$$\forall p \in P, D_{p,t}^-(\mu) \leq \#(p, \mu) \wedge (D_{p,t}^\circ(\mu) > \#(p, \mu) \vee D_{p,t}^\circ(\mu) = 0)$$

When transition t fires in marking μ the new marking μ' satisfies:

$$\forall p \in P, \#(p, \mu') = \#(p, \mu) - D_{p,t}^-(\mu) + D_{p,t}^+(\mu)$$

- $\forall t \in T, e_t : \mathbb{N}^{|P|} \rightarrow \{\text{true}, \text{false}\}$ is the enabling function of transition t . If $e_t(\mu) = \text{false}$, t is disabled in μ .

- $>$ is a transitive and irreflexive relation imposing a priority among transitions. In a marking μ , t_1 is marking-enabled iff it is arc-enabled, $e_{t_1}(\mu) = \text{true}$, and no other transition t_2 exists such that $t_2 > t_1$, t_2 is arc-enabled, and $e_{t_2}(\mu) = \text{true}$. This definition is more flexible than the one adopted by other SPN formalisms, where integers are associated with transitions (e.g., imagine the situation where $t_1 > t_2$, $t_3 > t_4$, but t_1 has no priority relation with respect to t_3 and t_4).
- μ_0 is the initial marking.
- $\forall t \in T, \lambda_t : \mathbb{N}^{|P|} \rightarrow \mathbb{R}^+ \cup \{\infty\}$ is the rate of the exponential distribution for the firing time of transition t . If the rate is ∞ in a marking, the transition firing time is zero. This is a generalization of [1], where transitions are *a priori* classified as “timed” or “immediate”. In this paper, though, there are transitions for which the rate is always ∞ . We still call them immediate and we represent them with a thin bar instead of a hollow rectangle. The distinction between vanishing and tangible markings introduced in [1] is still applicable: a marking μ is said to be vanishing if there is a marking-enabled transition t in μ such that $\lambda_t = \infty$; μ is said to be tangible otherwise. We additionally impose the interpretation that, in a vanishing marking μ , all transitions t with $\lambda_t(\mu) < \infty$ are implicitly inhibited. Hence, a transition t in a marking μ is enabled in the usual sense and can actually fire iff it is marking-enabled and either μ is tangible or μ is vanishing and $\lambda_t(\mu) = \infty$.
- $\forall t \in T, w_t : \mathbb{N}^{|P|} \rightarrow \mathbb{R}^+$ describes the weight assigned to the firing of enabled transition t , whenever its rate λ_t evaluates to ∞ . Assume that the set of transitions $X \subseteq T$ is enabled in a vanishing marking μ . Then, the probability of firing transition t in μ is given by $w_t(\mu) / (\sum_{v \in X} w_v(\mu))$. If a marking-dependent weight specification is not needed, the definition of w can be reduced to $\forall t \in T, w_t \in \mathbb{R}^+$.

The SRN components described so far define a trivariate discrete-parameter stochastic process: $\{(\mu_n, \tau_n, \theta_n), n \in \mathbb{N}\}$. μ_n is the n -th marking encountered, $\tau_n \in T$ is the n -th transition to fire (marking μ_{n+1} is obtained by firing transition τ_n in μ_n), and $\theta_n \geq 0$ is the time at which

it fires ($\theta_i \geq \theta_{i-1}$). It is also possible to define a continuous-time process describing the marking at time θ , $\{\mu(\theta), \theta \geq 0\}$, which is completely determined given $\{(\mu_n, \tau_n, \theta_n), n \in \mathbb{N}\}$: $\mu(\theta) = \mu_{\sup\{n: \theta_n \leq \theta\}}$. This process describes only the evolution with respect to the tangible markings, that is, $Pr\{\mu(\theta) \text{ is vanishing}\} = 0$.

The last component of an SRN specification defines the measures to be computed:

- $M = \{(\rho_1, r_1, \phi_1), \dots, (\rho_{|M|}, r_{|M|}, \phi_{|M|})\}$ is a finite set of measures, each specifying the computation of a single real value. A measure $(\rho, r, \phi) \in M$ has three components. The first and second components specify a reward structure over the underlying stochastic process $\{(\mu_n, \tau_n, \theta_n), n \in \mathbb{N}\}$. $\rho : \mathbb{N}^{|P|} \rightarrow \mathbb{R}$ is a reward rate: $\rho(\mu)$ is the rate at which reward is accumulated when the marking is μ . $\forall t \in T, r_t : \mathbb{N}^{|P|} \rightarrow \mathbb{R}$ is a reward impulse: $r_t(\mu)$ is the instantaneous reward gained when firing transition t while in marking μ . Often, a marking-dependent reward impulse specification is not needed and the definition of r can be simplified accordingly. The reward structure specified by ρ and r over $\{(\mu_n, \tau_n, \theta_n), n \in \mathbb{N}\}$ defines a new stochastic process $\{Y(\theta), \theta \geq 0\}$, describing the reward accumulated by the SRN up to time θ :

$$Y(\theta) = \int_0^\theta \rho(\mu(u)) du + \sum_{n: \theta_n \leq \theta} r_{\tau_n}(\mu_n)$$

The third component of a measure specification, ϕ , is a function that computes a single real value from the stochastic process $\{Y(\theta), \theta \geq 0\}$. If \mathcal{R} is the set of real-valued stochastic processes with index over the naturals, then $\phi : \mathcal{R} \rightarrow \mathbb{R}$. The generality of this definition is best illustrated by showing the wide range of measures the triplet (ρ, r, ϕ) can capture (in some SRNs, some of these measures might be infinite):

- Expected number of transition firings up to time θ : this is simply $E[Y(\theta)]$ when all reward rates are zero and all reward impulses are one.
- Expected time-averaged reward up to time θ : $E \left[\frac{Y(\theta)}{\theta} \right]$.
- Expected instantaneous reward rate at time θ : $E \left[\lim_{\delta \rightarrow 0} \frac{Y(\theta+\delta) - Y(\theta)}{\delta} \right]$.
- Expected accumulated reward rate in steady-state: $E [\lim_{\theta \rightarrow \infty} Y(\theta)]$.

- Mean time to absorption: this is a particular case of the previous measure, obtained by setting the reward rate of transient and absorbing states to one and zero, respectively, and all reward impulses to zero.
- Expected instantaneous reward rate in steady-state: $E \left[\lim_{\theta \rightarrow \infty} \lim_{\delta \rightarrow 0} \frac{Y(\theta+\delta) - Y(\theta)}{\delta} \right]$, which is also the same as the expected time-average reward in steady-state: $E \left[\lim_{\theta \rightarrow \infty} \frac{Y(\theta)}{\theta} \right]$.
- Supremum reward rate (assume that all reward impulses are zero):

$$\sup_{\theta \geq 0} \left\{ v : v \in \mathbb{R} \wedge Pr \left\{ \lim_{\delta \rightarrow 0} \frac{Y(\theta+\delta) - Y(\theta)}{\delta} = v \right\} > 0 \right\}.$$

This quantity can be expressed more simply using the stochastic process $\{(\mu_n), n \in \mathbb{N}\}$: $\sup_{n \geq 0} \left\{ \rho(\mu) : Pr\{\mu^{[n]} = \mu\} > 0 \right\}.$

Our intention is to define parametric SRNs. This can be accomplished by allowing each component of an SRN to depend on a set of parameters $\nu = (\nu_1, \dots, \nu_m) \in \mathbb{R}^m$:

$$\mathbf{A}(\nu) = \{P(\nu), T(\nu), D^-(\nu), D^+(\nu), D^o(\nu), e(\nu), >(\nu), \mu_0(\nu), \lambda(\nu), w(\nu), M(\nu)\}$$

Once the parameters ν are fixed, a simple (non-parametric) SRN is obtained.

The underlying stochastic process can be solved analytically to compute the probability of being in each tangible marking μ at time θ , $\pi_\mu(\theta)$, or in steady state, π_μ . It is also possible to directly compute the cumulative time spent in each tangible marking μ during the interval $[0, \theta]$, $\int_0^\theta \pi_\mu(\tau) d\tau$. All the measures described in this paper are expressed as expectations using reward rates only and they can be easily computed as a linear combination of the values of these probabilities or cumulative times.

3 Analysis of a producer-consumer tasking system

Consider a computer system where data items produced by N_p producers are consumed by N_c consumers. The exchange of items between the N_p producer tasks and the N_c consumer tasks is performed using one additional buffer task. A pseudo-Ada description of this system appears in Figure 1 [7].

The buffer task stores the incoming items into array Slots, having N_s positions, and it uses the integer variable FullSlots to keep track of the number of non-empty slots. Producer tasks cannot pass items to the buffer task when FullSlots is equal to N_s , and consumer tasks cannot retrieve items from the buffer task when FullSlots is equal to 0. The number of produced items cannot then exceed the number of consumed items plus N_s . A larger value for N_s can only attenuate the effect of temporary increases in the production or consumption rates, but these are equal in the long run.

The mechanism by which two Ada tasks synchronize and exchange data is the "rendezvous". Whenever a producer task has an item ready to pass, it issues an "entry call" to the buffer task (line 16). If the buffer task accepts this entry call, the rendezvous takes place, FullSlots is incremented, and the item is copied into array Slots (lines 35-37); similarly, a rendezvous with a consumer (lines 41-43) decrements FullSlots by one.

Each "entry" (lines 05 and 06) has an associated queue, where tasks making an entry call wait for a rendezvous. The presence of "guards" EnablePut and EnableGet (lines 34 and 40) inhibits the rendezvous at the guarded entry if the boolean condition is false (the guard is "closed"). Table I describes the value assumed by these boolean predicates based on three factors (presence of tasks in each of the two queues and value of variable FullSlots), for the five different policies discussed. When the buffer task is ready to rendezvous, the following cases can arise:

- Only one guard is open, but its associated queue is empty. This happens only when all the slots are full and no consumer is waiting, or all the slots are empty and no producer is waiting. A rendezvous cannot take place right away; the buffer task waits until a task joins the queue with the open guard.
- Both guards are open, but their associated queues are empty. A rendezvous cannot take place; the buffer task waits for the first task to join any of the two queues.
- Only one guard is open and its associated queue contains at least one task. A rendezvous with the first task in that queue takes place immediately.

```

01  $N_p$  : constant := number of producers;
02  $N_c$  : constant := number of consumers;
03  $N_s$  : constant := number of buffer slots;

04 task Buffer is
05     entry Put( Item : in data );
06     entry Get( Item : out data );
07 end Buffer;

08 task type Producer;
09 task type Consumer;

10 Producers : array ( 1 ..  $N_p$  ) of Producer;
11 Consumers : array ( 1 ..  $N_c$  ) of Consumer;

12 task body Producer is
13     Item : data;
14 begin
15     loop
16         Buffer.Put( Item );
17         statements  $S_p$ ;
18     end loop;
19 end Producer;

20 task body Consumer is
21     Item : data;
22 begin
23     loop
24         Buffer.Get( Item );
25         statements  $S_c$ ;
26     end loop;
27 end Consumer;

28 task body Buffer is
29     Slots : array ( 1 ..  $N_s$  ) of data;
30     FullSlots : Natural := 0;
31 begin
32     loop
33         select
34             when EnablePut =>
35                 accept Put( Item : in data ) do
36                     FullSlots := FullSlots + 1;
37                     Slots( FullSlots ) := Item;
38                 end Put;
39             or
40             when EnableGet =>
41                 accept Get( Item : out data ) do
42                     Item := Slots( FullSlots );
43                     FullSlots := FullSlots - 1;
44                 end Get;
45             end select;
46         statements  $S_b$ ;
47     end loop;
48 end Buffer;

```

Figure 1: Pseudo-Ada description of the producer-consumer system.

- Both guards are open and their associated queues both contain at least one task. A rendezvous with either the first producer or the first consumer in their respective queues takes place immediately.

Only the fourth case requires a choice between a rendezvous with a producer and a rendezvous with a consumer. In its definition, Ada makes no guarantee about which queue is actually selected. If a particular selection policy is desired, it can be enforced by modifying the guard predicates so that, when no queue is empty, exactly one guard is open.

In Table I, five different policies are presented:

- Nondeterministic (ND): nondeterministically select either a producer or a consumer, with uniform probability. This can be accomplished, for example, using a pseudo-random number generator. It is also possible to remember the selection made the previous time in this situation and toggle the selection; this is likely to be faster, but it introduces a correlation in the sequence of selections.
- Producer First (PF): select a producer.
- Consumer First (CF): select a consumer.
- Proportional (PR): nondeterministically select either a producer or a consumer, but, instead of using uniform probability for producers and consumers, use a probability split proportional to the number of empty and full slots, respectively. This bias tends to keep the number of empty and full slots more balanced, which is intuitively a good idea.
- Threshold (TH): choose a producer if more slots are empty than full; choose a consumer if more slots are full than empty; choose either with uniform probability if exactly half of the slots are full. This policy tries to achieve the same goal as the previous one, but deterministically. When exactly half of the slots are full, the behavior is the same as in the ND policy; for simplicity, we assume N_s to be odd, so this case cannot arise.

So far, the description of the system has been focused on the software, but the actual timing behavior is determined also by the hardware architecture and by the allocation of tasks

Table I
Values for Boolean Predicates EnablePut and EnableGet.

Policy	Condition			Value returned	
	Producers Waiting	Consumers Waiting	Value of FullSlots	EnablePut	EnableGet
X	X	X	0	true	false
X			N_s	false	true
X	no	no	$1..N_s - 1$	true	true
X	yes	no		true	X
X	no	yes		X	true
ND	yes	yes	$1..N_s - 1$	α	not α
PF	yes	yes	$1..N_s - 1$	true	false
CF	yes	yes	$1..N_s - 1$	false	true
PR	yes	yes	$1..N_s - 1$	β	not β
TH	yes	yes	$1..\lfloor N_s - 1 \rfloor / 2$	true	false
			$\lceil N_s + 1 \rceil / 2..N_s - 1$	false	true
			$N_s / 2 \quad (N_s \text{ even})$	α	not α
<p>Note: X means that the value is not relevant.</p> <p>α is a boolean random variable with $Pr\{\alpha = true\} = 1/2$.</p> <p>$\beta$ is a boolean random variable with $Pr\{\beta = true\} = (N_s - Fullslots)/N_s$.</p>					

to processors. Three possibilities are considered:

- SINGLE: a classic single processor architecture, where all tasks share the same CPU.
- THREE: a three-processor architecture, one processor for the N_p producer tasks, another

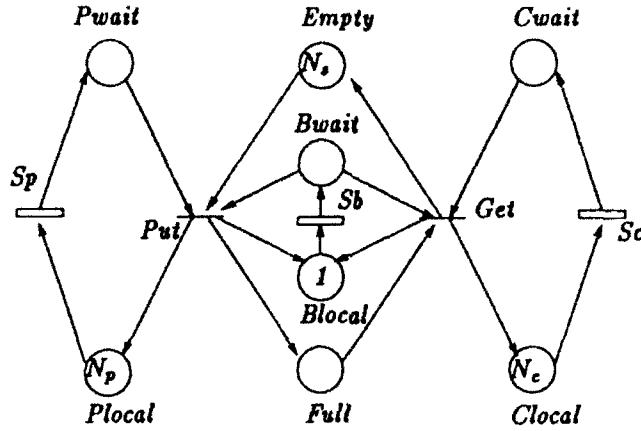


Figure 2: The SRN for the producer-consumer system.

for the N_c consumer tasks, and the last one for the single buffer task.

- MANY: a one-processor-per-task architecture, with no processor sharing.

The actual number of processors in the actual system could probably be somewhere between 3 and $N_p + N_c + 1$, the single processor architecture is considered mainly for reference.

3.1 SRN model

The system just described is concisely modeled by the SRN in Figure 2. Tokens in places *Plocal*, *Clocal*, and *Blocal* represent tasks (of the appropriate type) executing the statements *Sp*, *Sc*, and *Sb*, respectively, while tokens in places *Pwait*, *Cwait*, and *Bwait* represent tasks waiting for a rendezvous at the *Put* or *Get* entries. The tokens in places *Empty* and *Full* count the number of empty and full slots, respectively.

Transitions *Sp*, *Sc*, and *Sb* are assumed to be “black boxes” with an exponentially distributed time duration, but they could be changed into a more detailed phase-type expansion (using a “subnet”) if more information were available about the actual structure of the code. This would increase the size of the reachability graph, but it would also allow a more precise representation of the timing behavior in the case exponential distributions were not adequate.

Immediate transitions *Put* and *Get* correspond to the actions in the rendezvous (lines 34-38

and 40-44 in Figure 1, respectively), which are modeled as instantaneous, since the time spent for them is likely to be negligible compared to the other blocks of statements (if not, the SRN could be modified to represent these times durations explicitly).

The five different selection policies described in the previous section are obtained by defining the appropriate value for predicates EnabledPut and EnableGet. Exactly analogous to them, and even simpler to specify, are the "enabling functions" e_{Put} and e_{Get} associated with transitions *Put* and *Get*, respectively:

$$e_{Put} = \begin{cases} false & \text{if (policy = CF and } \#(Cwait) > 0 \text{ and } \#(Full) > 0) \\ & \text{or (policy = TH and } \#(Cwait) > 0 \text{ and } \#(Empty) > \#(Full)) \\ true & \text{otherwise} \end{cases}$$

$$e_{Get} = \begin{cases} false & \text{if (policy = PF and enabled(Put)) \\ & \text{or (policy = TH and } \#(Pwait) > 0 \text{ and } \#(Empty) < \#(Full)) \\ true & \text{otherwise} \end{cases}$$

In addition, the probabilistic choices in the the ND and PR policies (and TH, when N_s is even) can be specified by assigning weights w_{Put} and w_{Get} to the two transitions:

$$w_{Put} = \begin{cases} \#(Empty) & \text{if policy = TH} \\ 1 & \text{otherwise} \end{cases}$$

$$w_{Get} = \begin{cases} \#(Full) & \text{if policy = TH} \\ 1 & \text{otherwise} \end{cases}$$

The specification of the rates for the remaining three transitions completes the description of the SRN. These rates are related to the times required to execute the blocks of statements *Sp*, *Sc*, and *Sb*, but also to the type of hardware architecture, since sharing the processor slows down the execution. Table II shows the firing rates used assuming perfect processor sharing with no context switch overhead, and assuming that the times required to execute blocks *Sp*, *Sc*, and *Sb* for a task running on a processor in isolation (no sharing) are 0.003, 0.003, and 0.0005 seconds, respectively.

Table II

Rates for the Transitions of the Producer-Consumer SRN

Transition	Architecture	Firing rate (sec^{-1})
λ_{Sp}	SINGLE	$\#(P_{local}) / (0.003(\#(P_{local}) + \#(C_{local}) + \#(B_{local})))$
	THREE	$1/0.003$
	MANY	$\#(P_{local}) / 0.003$
λ_{Sc}	SINGLE	$\#(C_{local}) / (0.003(\#(P_{local}) + \#(C_{local}) + \#(B_{local})))$
	THREE	$1/0.003$
	MANY	$\#(C_{local}) / 0.003$
λ_{Sb}	SINGLE	$1 / (0.0005(\#(P_{local}) + \#(C_{local}) + 1))$
	THREE	$1/0.0005$
	MANY	$1/0.0005$

Before concluding this section, it is useful to compute the number of markings generated by the SRN analysis, both to check the correctness of the model and to avoid attempting solutions that require excessive resources (memory in particular). For the parametric SRN of Figure 2, this number is a function of the parameters N_p , N_c , and N_s (the policy and the architecture are also parameters, but they do not affect the number of markings). Table III shows how to count the exact number of vanishing and tangible markings using a case analysis. Since the number of markings grows as $O(N_s N_p N_c)$, it is possible to study the system for reasonably large values of these three parameters (SRNs with $\approx 10^4$ markings can be normally analyzed in a few minutes on a workstation, but SRNs with $\approx 10^5$ or even $\approx 10^6$ markings can be solved in a matter of hours, if enough memory is available).

Table III
Marking Count for the Producer-Consumer SRN

Contents of place				Markings	
<i>Empty</i>	<i>Pwait</i>	<i>Cwait</i>	<i>Bwait</i>	Type	Count
$0..N_s$	$0..N_p$	$0..N_c$	0	tangible	$(N_s + 1)(N_p + 1)(N_c + 1)$
$0..N_s$	0	0	1	tangible	$N_s + 1$
$0..N_s - 1$	0	$1..N_c$	1	vanishing	$N_s N_c$
$1..N_s$	$1..N_p$	0	1	vanishing	$N_s N_p$
$0..N_s$	$1..N_p$	$1..N_c$	1	vanishing	$(N_s + 1)N_p N_c$
N_s	0	$1..N_c$	1	tangible	N_c
0	$1..N_p$	0	1	tangible	N_p
tangible markings: $(N_s + 1)((N_p + 1)(N_c + 1) + 1) + N_p + N_c$ vanishing markings: $(N_s + 1)N_p N_c + N_p(N_p + N_c)$					

3.2 Performance analysis

The five policies defined earlier have a simple implementation in Ada. Even the ones requiring a pseudo-random number generator introduce only a small overhead compared to the number of statements likely to constitute the blocks Sp , Sc , and Sb .

The selection of a policy among the five ones presented could then be based on the effect that these policies have on the performance of the system (in steady-state). Different aspects of the system behavior might be the most relevant in defining "performance":

- Response time for producers, consumers, or both.
- Probability distribution of the number of producers blocked because all slots are full.
- Probability distribution of the number of consumers blocked because all slots are empty.

- Throughput of the system, expressed as the number of items passed from any producer to any consumer in a unit of time.

For the purpose of this study, the throughput of the system, τ , is used. In the SRN of Figure 2, the throughput of the producers, τ_p , can be computed by defining the reward rate in marking μ as $\lambda_{Sp}(\mu)$, the rate of transition Sp , and computing the expected reward rate in steady-state:

$$\tau_p = \sum_{\mu} \lambda_{Sp}(\mu) \pi_{\mu}$$

τ_c and τ_b can be computed in a similar way, or by observing that $\tau = \tau_p = \tau_c = \tau_b/2$.

The value of τ with the SINGLE architecture is the same independent of the policy adopted and of the number of slots, N_s , producers, N_p , or consumers, N_c (as long as none is zero): $\tau = 142.857 \text{ sec}^{-1}$. The reason is that the only processor is always busy, so τ is simply the inverse of the total time spent to process each item: 0.003 seconds in the producer task, 0.0005 seconds in the buffer task after the rendezvous with a producer, plus another 0.0005 seconds after the rendezvous with a consumer, and finally 0.003 seconds in the consumer task, for a total of 0.007 seconds ($1/0.007 = 142.857$).

With the THREE and MANY architectures, τ is instead affected by the three parameters. Figures 3 and 4 plot τ as a function of N_s for these two architectures using the ND policy in a balanced system ($N_p = N_c$). The effect of different policies is minor compared to doubling N_p and N_c , so it is studied later in this section.

With the THREE architecture, the improvement due to the increase in N_s is sublinear. In addition, it is less noticeable for larger values of $N_p = N_c$, since, after a certain point, the processors for the producers and the consumers become the bottleneck. In this case, the limit for τ is the inverse of the maximum of the time spent on each item by each processor (0.003, 0.001, and 0.003 sec respectively): $\lim_{N_p, N_c, N_s \rightarrow \infty} \tau = 1/0.003 \text{ sec}^{-1} = 333.333 \text{ sec}^{-1}$. For example, $\tau = 329.340 \text{ sec}^{-1}$ when $N_p = N_c = 32$ and $N_s = 19$ (not shown).

The improvement due to increasing N_s with the MANY architecture is even smaller. Furthermore, it appears that the system is saturated when $N_p = N_c = 8$ and no appreciable improvement is achieved by increasing N_s . The reason is again to be found in the bottleneck,

this time the buffer task. The production (and consumption) rate could now be as high as $N_p/0.003 \text{ sec}^{-1}$, since each task has a dedicated processor, but the buffer task is involved in two rendezvous for each item, so that an upper bound on τ is given by $1/0.001 = 1000 \text{ sec}^{-1}$. Since $4/0.003 > 1000$, it appears that the buffer task is already the bottleneck when $N_p = N_c = 4$, although, in this case, increasing the number of slots still has a visible effect on τ (but the increase is smaller than when $N_p = N_c = 1, 2$). To summarize this first part of the analysis:

- It is advantageous to increase the number of producers and consumers, *even if the total computational capacity remains constant* (architecture THREE). Depending on the nature of the system, though, this may not be possible, since the number of tasks could be dictated by external considerations (e.g., each producer task monitors a different sensor).
- Increasing the number of slots is always advantageous, but particularly so when only a few producer and consumer tasks are present.
- On a highly parallel architecture (MANY), the buffer task soon becomes a bottleneck. This points out a limitation of the Ada rendezvous. The buffer task must be introduced because, in Ada, a task performing an entry call must know the identity of the callee, so it is not possible to let a producer rendezvous directly with *any* consumer using a single entry call. This problem can be alleviated by having several buffer tasks and partitioning the producer and consumer tasks so that each buffer task serves only a subset of the producers and consumers. Partitioning, though, introduces a different kind of inefficiency. Producers associated to a buffer task having all the slots full sit idle, even if other buffer tasks may have some or even all the slots empty.

Considering now the effect of the selection policies, it is immediately apparent that there is no absolute "optimal" policy. Figure 5 shows τ as a function of N_s in an unbalanced system ($N_p = 4, N_c = 2$), with the MANY architecture, for the five policies. The ability of producers to produce is higher than the ability of consumers to consume, hence giving precedence to consumers (CF policy) tends to restore the balance and is the optimal choice, while the PF policy increases the unbalance, resulting in the worst throughput. The plot for $N_p = 2, N_c = 4$

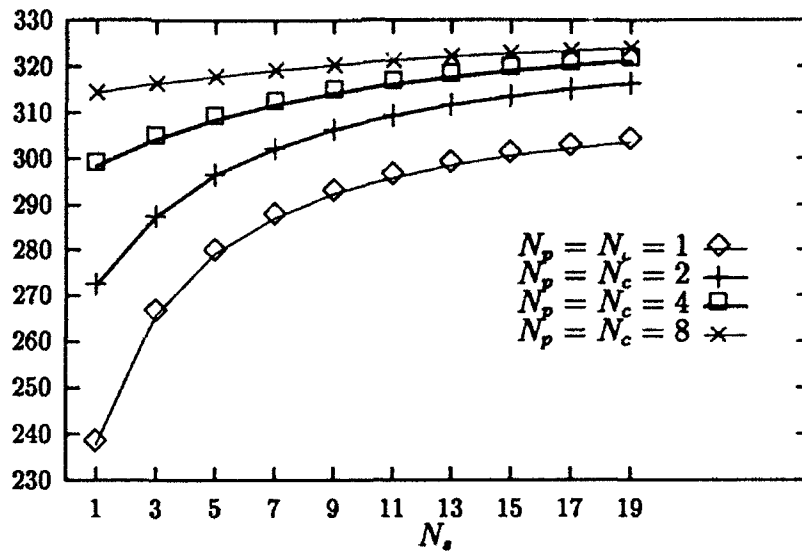


Figure 3: τ (in sec^{-1}) as a function of N_s (THREE architecture, ND policy).

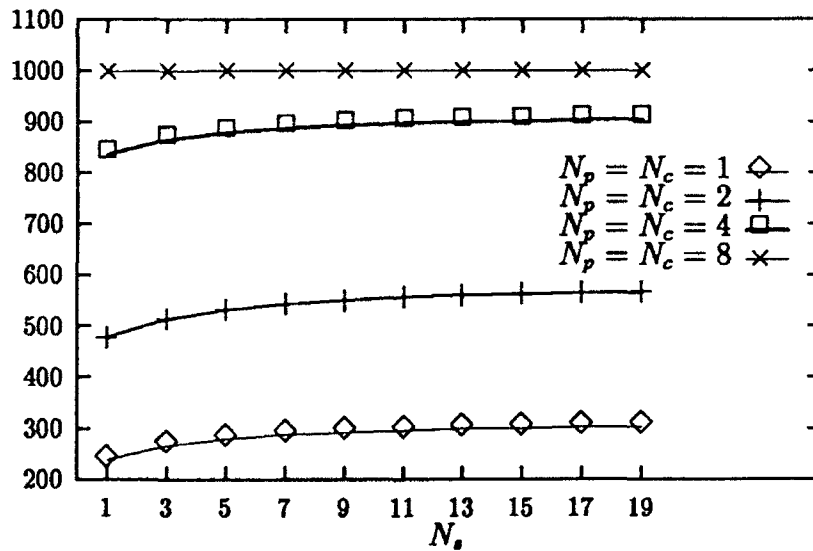


Figure 4: τ (in sec^{-1}) as a function of N_s (MANY architecture, ND policy).

(not shown) is exactly the same as the one for $N_p = 4$, $N_c = 2$, with the exception that the labels for the PF and CF policies are reversed: the PF policy is now optimal while the CF policy is the worst, and the others achieve the same value (the ND, PR, and TH policies are

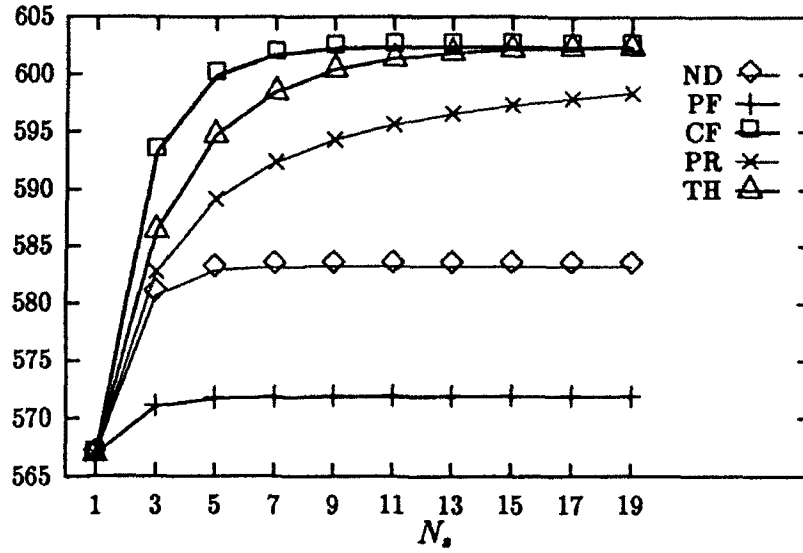


Figure 5: τ (in sec^{-1}) as a function of N_s (MANY architecture, $N_p = 4$, $N_c = 2$).

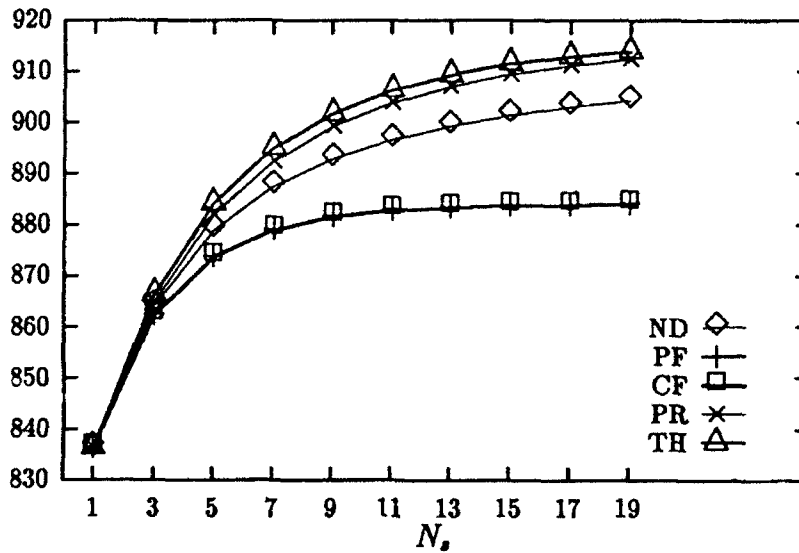


Figure 6: τ (in sec^{-1}) as a function of N_s (MANY architecture, $N_p = 4$, $N_c = 4$).

“symmetric”, so it should not be surprising that they result in the same throughput when $N_p = 4$, $N_c = 2$ and when $N_p = 2$, $N_c = 4$).

The TH policy is the second best in either case, followed by the PR and ND policies, in that

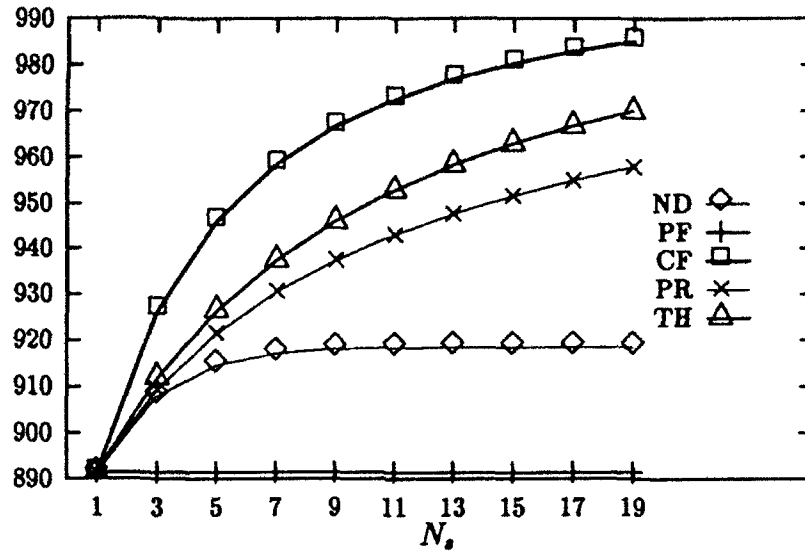


Figure 7: τ (in sec^{-1}) as a function of N_s (MANY architecture, $N_p = 8$, $N_c = 4$).

order. This can be justified by observing that the PR policy is a (not so useful) compromise between the TH policy, which deterministically tries to achieve balance in the number of used and free slots, and the ND policy, which completely ignores the status of the slots.

The same reasoning explains the effect of the five policies in a balanced system where $N_p = N_c = 4$ (Figure 6). The two asymmetrical policies, PF and CF, are equally poor, while the TH, PR, and ND policies are at the top, in that order.

The TH policy is a consistently good choice (nearly optimal in an unbalanced system, optimal in a balanced system). If the number of producers and consumers is subject to change during the deployment of the system, the TH policy is the best choice because of its easy implementation and predictable performance. For example, a system initially unbalanced in favor of consumers could suggest the adoption of the PF policy, but inefficiencies would arise if the situation had to be reversed later (the difference between the best and worst policy in Figure 5 is over 5%).

The plots for the PF and CF policy in Figure 6 appear to have a much slower rate of increase than the plots for the other three policies. This can be explained by considering

what happens with the PF policy when the system is not unbalanced toward the consumers ($N_p \geq N_c$) and the buffer task is the bottleneck ($N_c/0.003 > 1/0.001$). In this case, there are often both producers and consumers waiting whenever the buffer is ready to rendezvous. The PF policy, though, chooses a producer whenever possible, that is, whenever there is at least one empty slot. The effect of this behavior, in the limit, is to let the number of full slots alternate between N_s (choose a consumer) and $N_s - 1$ (choose a producer), with a negative effect: the system might as well have just a single slot ($N_s = 1$), since the "window" between the number of produced and consumed items is effectively restricted to one most of the time.

Figure 7 is even more dramatic, showing no appreciable increase at all for the PF policy. The values of τ for a system with the MANY architecture, $N_p = 8$, $N_c = 4$, and $N_s = 125$ (not shown) confirm this observation. The difference between the optimal CF policy and the TH policy is minimal (994.3 sec^{-1} and 994.1 sec^{-1} , respectively), while the PF policy lags seriously behind (891.4 sec^{-1}). Even more illuminating is the inspection of the probabilities Π_e that all slots are empty and Π_f that all slots are full:

$$\Pi_e = \sum_{\mu: \#(Empty, \mu) = N_s} \pi_\mu = \begin{cases} < 10^{-12} & \text{(ND, PF, PR, and TH policies)} \\ 0.13929526 & \text{(CF policy)} \end{cases}$$

$$\Pi_f = \sum_{\mu: \#(Full, \mu) = N_s} \pi_\mu = \begin{cases} 0.275976 & \text{(ND policy)} \\ 0.553947 & \text{(PF policy)} \\ 0.000003 & \text{(CF policy)} \\ 0.010870 & \text{(PR policy)} \\ 0.000499 & \text{(TH policy)} \end{cases}$$

These probabilities should be kept small, since, when all slots are full (empty), no rendezvous can take place with a producer (consumer), thus increasing the probability that the buffer task, which is the real bottleneck, remains idle. While Π_e is numerically negligible only for the non-optimal policies, Π_f is the most relevant quantity to observe, since $N_p > N_c$; Π_f is small for both the CF and TH policies (about 170 times smaller for CF than for TH, although this

difference affects the actual throughput τ only marginally), but it is definitely high for the PF policy, resulting in a considerably smaller value of τ .

The average number ν_f of full slots also confirms this behavior:

$$\nu_f = \sum_{\mu} \#(Full, \mu) \pi_{\mu} = \begin{cases} 123.22 & \text{(ND policy)} \\ 124.55 & \text{(PF policy)} \\ 9.66 & \text{(CF policy)} \\ 100.72 & \text{(PR policy)} \\ 71.40 & \text{(TH policy)} \end{cases}$$

The TH policy does indeed achieve the value of ν_f closest to $N_s/2 = 62.5$, but is still sub-optimal, suggesting that the ideal value for ν_f is actually a function of N_p and N_c as well.

4 Fault-tolerant software

Design diversity as a means of achieving fault-tolerance in software has been suggested by several authors. Fault-tolerant software using this method include N-version programming [4] and recovery blocks [22]. The former uses voting on the results of various versions for error detection and the latter uses an acceptance test (AT) and rollback recovery. While these are the two major approaches to software fault-tolerance, several hybrid methods have also been proposed [22, 24].

In this section, we analyze the recovery block (RB) scheme; a fault-tolerant software construct that uses design diversity [22]. It consists of a primary module, one or more alternate modules and an AT. The primary and the alternate modules are based on different algorithms for the same problem and may be implemented by different programmers. On a given set of data inputs, the primary is executed first and the results are checked using the AT. Should the AT fail to accept the results, the alternate modules are invoked in succession until one is found to produce results that are accepted by the test or until all of them fail to satisfy the AT. In the latter case, the RB is said to have failed on this input data set. The pseudocode for a RB with a primary module and m alternate modules is shown below:

```

ensure acceptance test
by primary module
else by alternate module 1
else by alternate module 2
...
else by alternate module m
else error

```

Probabilistic models of RBs have been considered by several authors [3, 9, 25]. Discrete time Markov chains (DTMCs) have been used to derive measures like the probability of RB failure or the number of inputs (correctly) processed until RB failure; continuous time Markov chains (CTMCs) have been used to derive time-based measures like the mean time to failure (MTTF) or the (un)reliability of the RB. We note that if we are able to analyze a CTMC for transient cumulative measures besides transient instantaneous measures, we can derive both the above types of measures using a CTMC, i.e., we do not need to resort to two different formalisms depending on the measures desired.

Pucci [21] points out some of the difficulties in estimating the parameters used in earlier models. He classifies events occurring in a RB into four distinct categories based on the behavior of the alternate modules and the AT. Four different events can occur:

- (1) Module i produces correct results which the AT accepts.
- (2) Module i produces correct results which the AT rejects.
- (3) Module i produces incorrect results which the AT rejects.
- (4) Module i produces incorrect results which the AT accepts.

It is easier to estimate parameters corresponding to these events. We consider a similar event classification in the model presented in the next section.

4.1 SRN model

In this section, we present a general SRN model for the recovery block scheme. The primary module is indexed by 0 and the alternate modules are indexed 1 through m . The execution time of module i is assumed to be exponentially distributed with mean $1/\lambda_{Tm_i}$, and that of the AT is exponentially distributed with mean $1/\lambda_{Tatne_i} = 1/\lambda_{Tate_i}$. The probability that module i produces incorrect output is ρ_i . We assume that the AT fails to detect erroneous module output with probability p_e . This probability corresponds to event (4) mentioned above. We assume that this event is not catastrophic, unlike the assumption used by Pucci [21]. However, it is easy to change our model to make this event catastrophic. The AT might raise a false alarm with probability p_f , which corresponds to event (2) above. We assume that this event does not result in subsequent rejection of results from the other alternate modules, unlike as assumed by Pucci [21]. This assumption can easily be changed in the model. We let p_c be the probability that recovery following a failure to satisfy the AT is successful. We must realize that all the above probabilities pertaining to any module i are conditional probabilities, conditioned upon the fact that module i is actually invoked and that the previous $i-1$ modules have failed. Thus, the correlation between the software modules is automatically accounted for by the conditional nature of these probabilities.

The SRN model of a recovery block is shown in Figure 8. The net is nearly self-explanatory. Place Pm_0 is the starting point of the RB. The firing of transition Tm_0 corresponds to the completion of the execution of the primary module. Transitions Tne_0 and Te_0 correspond to the events that the results produced by the module are correct and incorrect respectively and have weights $1 - \rho_i$ and ρ_i , respectively. Transition $Tatne_0$ represents the execution of the AT after the module produces correct results. The immediate transitions Ts_0 and Tfa_0 , which correspond to events (1) and (2) mentioned above, are then enabled. The weights of these two transitions are given by $1 - p_f$ and p_f respectively. Transition $Tate_0$ represents the execution of the AT after the module produces incorrect results. The immediate transitions Tse_0 and Tee_0 , which correspond to events (3) and (4) mentioned above, are then enabled. The weights of these two transitions are $1 - p_e$ and p_e respectively. Once an error is discovered, represented

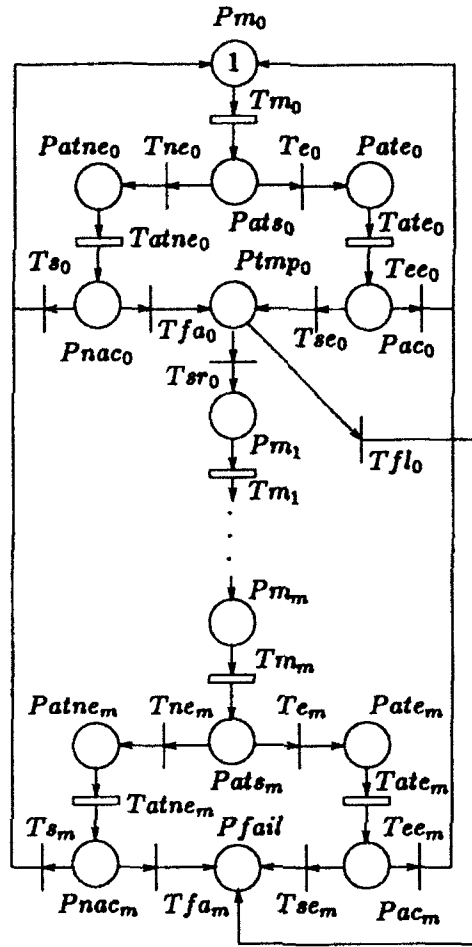


Figure 8: SRN model for a recovery block.

by the firing of either Tfa_0 and Tse_0 , the system initiates a recovery action. Transition Tsr_0 represents a successful recovery after a failure of the AT and transition Tfl_0 represents an unsuccessful recovery, thus resulting in the RB failure. The corresponding weights of these two transitions are p_c and $1 - p_c$ respectively. The output arc from Tsr_0 leads to Pm_1 , the starting place of the first alternate module, while the output arc from Tfl_0 leads to $Pfail$ which represents the RB failure.

The alternate modules are similarly modeled by the other places and transitions indexed from 1 to m . The structure of the last module is slightly different, since the failure of the last module automatically results in a system failure. Thus, the output arcs from transitions Tfa_m

and Tse_m lead to place $Pfail$.

We can compute the mean time to recovery block failure or the distribution of time to recovery block failure (its complement is the reliability of recovery block). For this purpose, we assign reward rate 1 to all markings in which there is no token in place $Pfail$; all other markings are assigned a reward rate equal to zero. If we now compute the expected accumulated reward until system failure, we obtain the $MTTF$:

$$MTTF = \sum_{\mu: \#(Pfail, \mu)=0} \int_0^{\infty} \pi_{\mu}(\tau) d\tau$$

By computing the expected reward rate at time θ , we obtain instead RB reliability at time θ :

$$R(\theta) = \sum_{\mu: \#(Pfail, \mu)=0} \pi_{\mu}(\theta)$$

The unreliability $UR(\theta)$, or probability of being failed by time θ , is simply given by $1 - R(\theta)$.

We can also compute the number of number of data sets processed until system failure, N , or until a specified time θ , $N(\theta)$. To compute these quantities, we assign the rate of transition Tm_0 , $\lambda_{Tm_0}(\mu)$, as the reward of marking μ . The expected accumulated reward until system failure or by time θ yield respectively N and $N(\theta)$.

$$N = \sum_{\mu} \lambda_{Tm_0}(\mu) \int_0^{\infty} \pi_{\mu}(\tau) d\tau$$

$$N(\theta) = \sum_{\mu} \lambda_{Tm_0}(\mu) \int_0^{\theta} \pi_{\mu}(\tau) d\tau$$

The $MTTF$ and N , the expected number of inputs processed until system failure, as a function of the number of modules available in the system (including the primary module) are shown in Figure 9. We assume that the execution rate of the primary module is $\lambda_{Tm_0} = 1 \text{ min}^{-1}$. For each alternate module, we assume that the execution rate is three-quarters that of the previous module, i.e., $\lambda_{Tm_i} = 0.75\lambda_{Tm_{i-1}}$. This is a reasonable assumption, since we would tend to use the fastest module as the primary module. The execution rate of the AT is $\lambda_{Tate_i} = \lambda_{Tatne_i} = 100 \text{ min}^{-1}$. The probabilities are $\forall i, \rho_i = 0.1, p_e = 0.01, p_f = 0.01$, and $p_c = 0.999$. Figure 9 shows how an increase in the number of alternate modules causes an increase in $MTTF$ and N . Further, it is interesting to note that the greatest benefit of

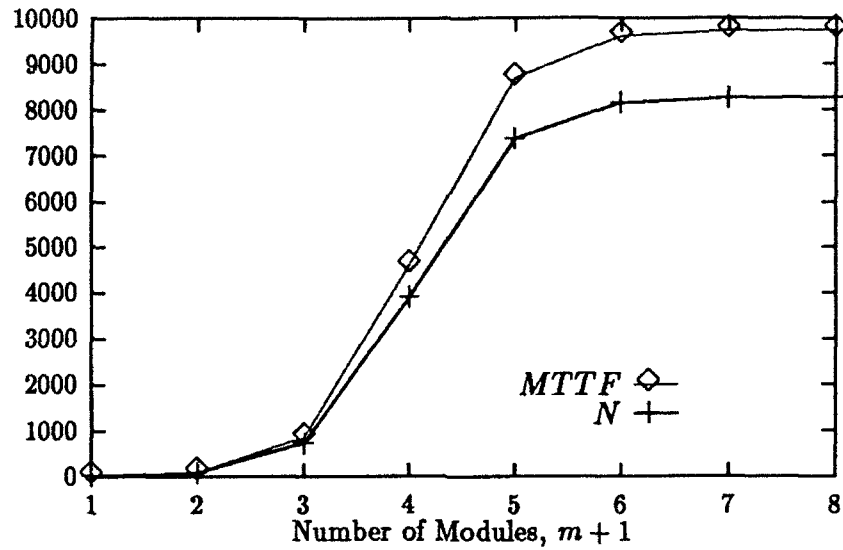


Figure 9: *MTTF* and *N* for the RB as function of the number of modules.

increasing the number of alternate modules is between 1 and 5. Beyond five alternate modules, the additional benefit is quite small.

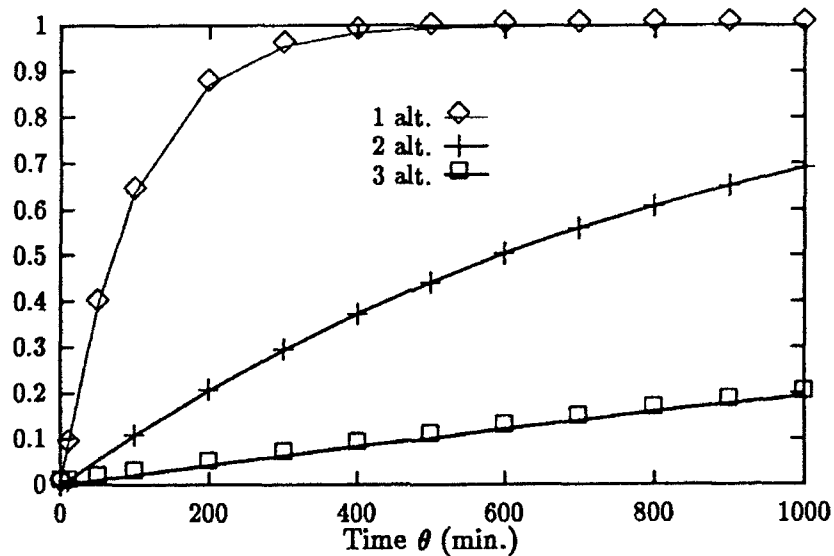


Figure 10: Failure probability for the RB as a function of time.

The distribution of the time to failure for the RB with 1, 2, and 3 alternate modules is

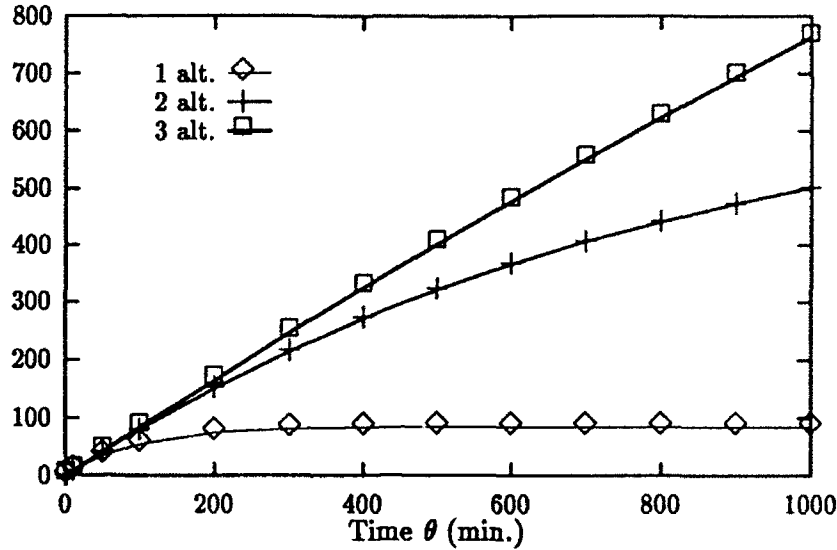


Figure 11: Number of inputs processed by the RB as a function of time.

plotted in Figure 10. From the figure we can see that the probability that the RB has failed by any given time θ decreases with increase in the number of alternate modules. The number of inputs processed by time θ , $N(\theta)$, for the RB with 1, 2, and 3 alternate modules is shown in Figure 11. The value of $N(\theta)$ levels off beyond $\theta = 400$ min for the system with 1 alternate, since the RB is very likely to have failed by that time.

4.2 Extensions

4.2.1 Clustering in the input data stream

The failure points in the input space for the RB tend to occur in clusters [2]. The sequence of input values to the RB tend to change slowly with time, thus, given a failure of the primary module for a given input, there is a greater likelihood of it failing for subsequent inputs. This clustering behavior in the input data stream should be taken into account. Csenki [9] considers a discrete time Markov model of a RB with failure clustering. He assumes that the system has a primary module and a single alternate. Given that the primary module has failed for a particular input, the number of subsequent inputs for which the module fails is assumed to be

a random variable ξ . The length of this additional sequence is however upper-bounded by a fixed value σ . Thus, we can define the probabilities $p_i = Pr\{\xi = i\}$ where $0 \leq i \leq \sigma$.

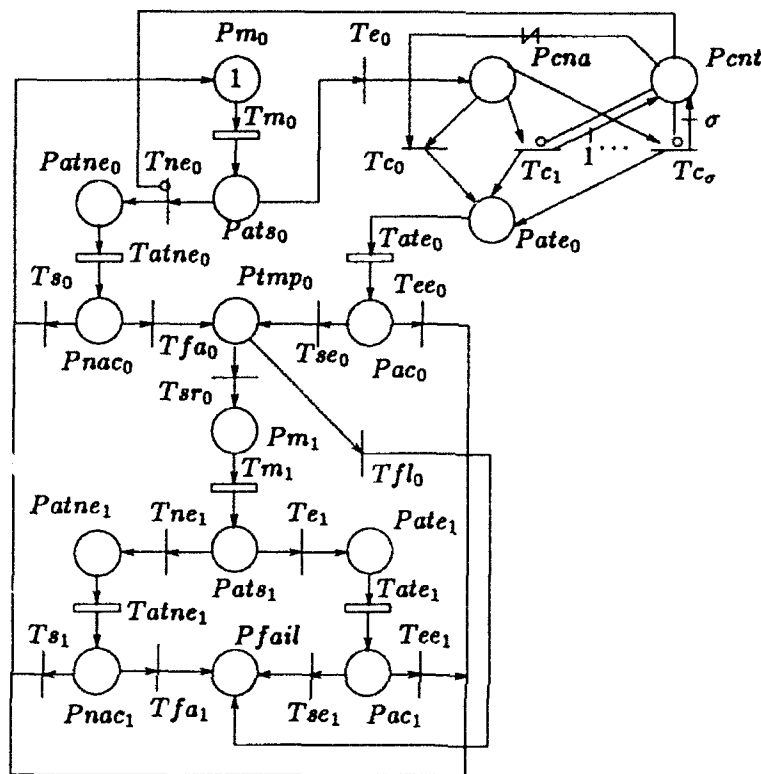


Figure 12: SRN model for a RB with clustered failures.

A SRN model for the RB with clustered failures and $m = 1$ is shown in Figure 12. We assume that the system has one primary module and a single alternate. The structure of this SRN is similar to that of the original SRN in Figure 8. The additional places P_{cna} and P_{cnt} together with the transitions $T_{c_0}, T_{c_1}, \dots, T_{c_\sigma}$, model the clustering in the input space. Transitions $T_{c_0}, \dots, T_{c_\sigma}$ correspond to the events where the size of the input cluster is given by $0, \dots, \sigma$ respectively. When the first datum of the input sequence that causes a clustered failure is encountered, it causes a failure of the primary module. Thus, the immediate transition T_{e_0} fires, depositing a token in place P_{cna} . Then immediate transitions $T_{c_0}, \dots, T_{c_\sigma}$ become enabled. The weights of these transitions are given by p_0, \dots, p_σ respectively. Whenever transition T_{c_i} fires, representing the fact that the primary module will fail for the next i

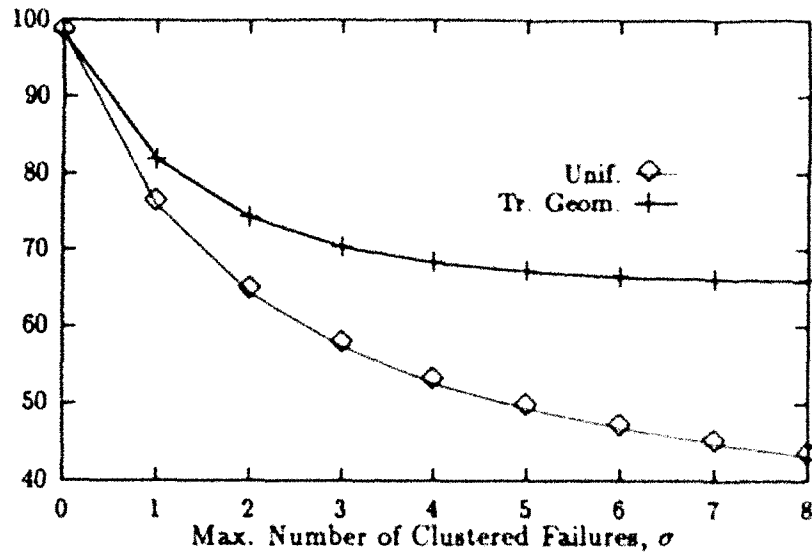


Figure 13: *MTTF* for the RB with clustered failures as a function of σ .

$0 \leq i \leq \sigma$, successive inputs, i tokens are deposited in place *Pcnt*, due to the multiple output arcs from transitions Tc_i to place *Pcnt*. At this point, transitions $Tnc_0, Tc_1, \dots, Tc_\sigma$ are disabled by the inhibitor arcs from *Pcnt*. Thereafter, for the next i times that Tm_0 fires, transition Te_0 will fire depositing a token in *Pcna*. Transition Tc_0 is now used to remove a token from *Pcnt* and starting the usual RB sequence corresponding to the case where the first module generates an erroneous output (token in *Pate_0*). This firing sequence continues until *Pcnt* is empty. To achieve this behavior, the input arc from *Pcnt* to Tc_0 has multiplicity 1 if $\#(Pcnt) \geq 1$ and 0 otherwise.

The *MTTF* as a function of σ is plotted in Figure 13 (the case with $\sigma = 0$ corresponds to the RB with no clustered failures). Two different distributions are considered for p_i , uniform and truncated geometric. For the uniform distribution, $\forall i, 0 \leq i \leq \sigma, p_i = 1/(\sigma + 1)$. In this case, given that a failure has occurred, the probability that the next i inputs also result in a failure of the primary is the same for all i (a pessimistic assumption). For the truncated geometric distribution, $\forall i, 0 \leq i \leq \sigma, p_i = p(1 - p)^i / (1 - (1 - p)^{\sigma+1})$, where $0 < p < 1$ (in Figure 13, we set $p = 0.5$). The probability that the failure cluster has size i tapers off as

i increases. This is more realistic than the uniform distribution. Clustered failures have a negative impact on the reliability of the recovery block. The effect is larger with the uniform distribution than with the truncated geometric distribution.

4.2.2 Arrivals of requests

So far, we have assumed that when the RB completes processing an input, another input value is already waiting to be processed. We are thus assuming a 100 % utilization of the recovery block. In reality, the input values usually arrive at random and are processed when the RB is available. To incorporate this effect into the model, we assume a Poisson input arrival process with rate λ_{Tarr} . We also assume that the RB has a finite buffer of size N ; at any time, no more than N inputs can be waiting for processing, including the one being processed. This is implemented by adding two places, P_{env} and P_{buf} , and a transition T_{arr} , resulting in Figure 14. The firing of transition T_{arr} represents the arrival of an input datum for processing. Whenever an input value successfully completes execution or escapes error detection, a token is added back into place P_{env} thus freeing up a buffer.

Since the RB cannot fail while it is not being used, taking into account the input arrival process increases the time to failure. This is reflected in Figure 15, where the $MTTF$ is plotted as a function of the arrival rate λ_{Tarr} for the RB with 1, 2, and 3 alternate modules, and $N = 10$. When λ_{Tarr} is very small, we notice that the $MTTF$ is large. This is because there is a greater probability of the RB being unused for longer periods of time. As λ_{Tarr} increases, the $MTTF$ approaches that of the basic system considered in the earlier section; in fact $\lambda_{Tarr} = \infty$ corresponds to this system. This is understandable since, when λ_{Tarr} increases, there is a greater chance of finding an input datum waiting for processing when the RB completes processing an earlier input value.

4.3 Other extensions

The software recovery block is executed on some form of hardware platform. In the earlier sections, we have implicitly assumed that the processor(s) on which the RB is being executed is

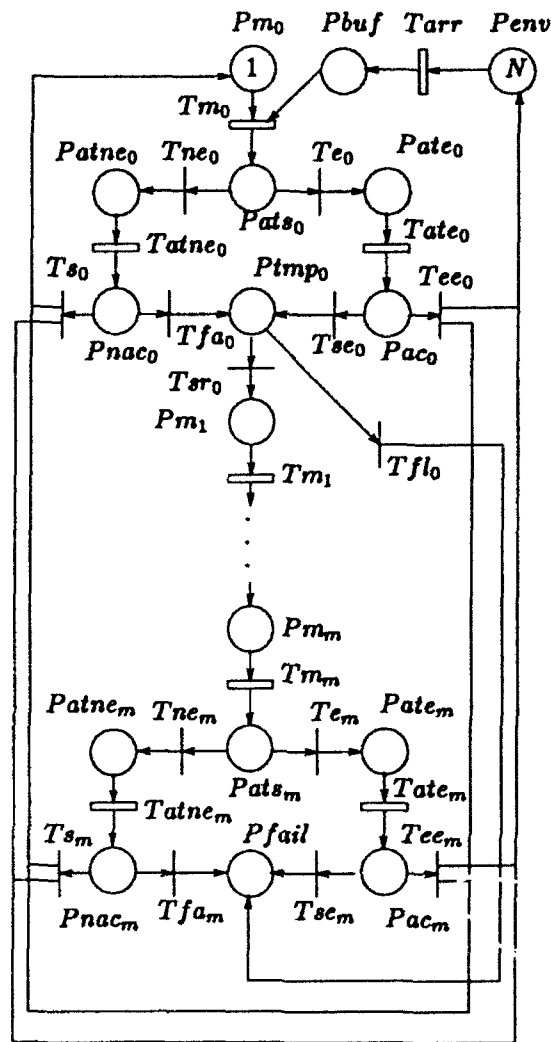


Figure 14: SRN model for a RB with input arrivals.

inherently fault free. In reality, hardware is subject to failures and can sometimes be repaired. Hence, any realistic model should take into account the behavior and characteristics of the underlying hardware such as processors and memory limitations. It is easy to extend our models to allow for the failure/repair behavior of the processor(s) or other hardware components. This will then allow us to carry out the combined evaluation of hardware and software.

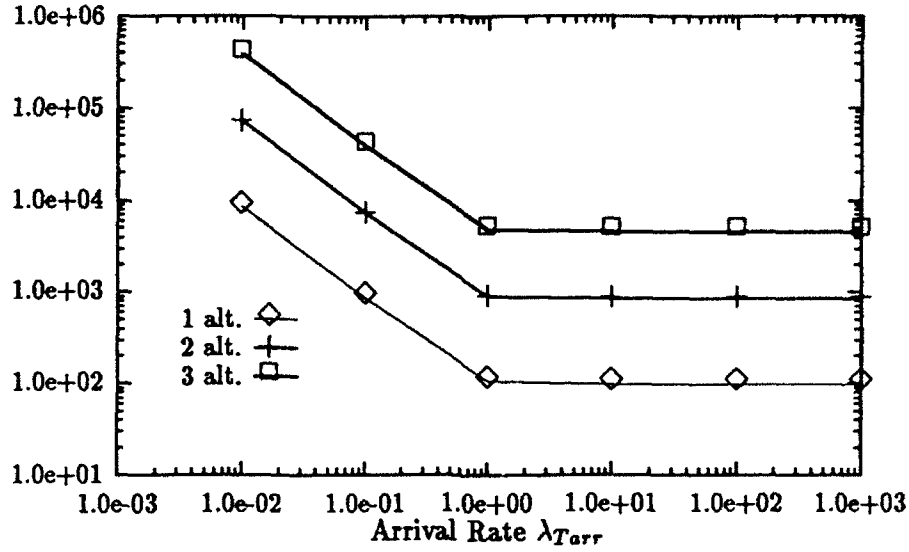


Figure 15: *MTTF* for a RB with input arrivals as a function of λ_{Tarr} .

5 Conclusions

In this paper, we presented two software modeling applications where SRNs can be effectively used to gain insight into a problem. The first application considers a producer-consumer tasking system with an intermediate buffer task, and studies how the performance is affected by different selection policies when multiple tasks are ready to synchronize.

The second application studies the reliability of a recovery block scheme. The initial model is incrementally augmented by considering the possibility of clustered failures or by taking into account the effective arrival rate of inputs to be processed by the system.

In either model, each quantity to be computed is defined in terms of either the expected value of a reward rate in steady-state or at a given time θ , or as the expected value of the accumulated reward until absorption or until a given time θ . This allows extreme flexibility while maintaining a rigorous formalization of these quantities.

References

- [1] Ajmone Marsan, M., Balbo, G., and Conte, G. A class of Generalized Stochastic Petri Nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems*. 2, 2 (May 1984), 93-122.
- [2] Ammann, P. E., and Knight, J. C. Data diversity: An approach to software fault tolerance. In *Proc. Seventeenth Int. Symp. on Fault-Tolerant Computing*. IEEE Computer Society Press, Los Alamitos, CA, July 1987, pp. 122-126.
- [3] Arlat, J., Kanoun, K., and Laprie, J. C. Dependability evaluation of software fault-tolerance. In *Proc. Eighteenth Int. Symp. on Fault-Tolerant Computing*. IEEE Computer Society Press, Los Alamitos, CA, June 1988, pp. 142-147.
- [4] Avizienis, A. The N-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.* SE-11, 12 (Dec. 1985), 1491-1501.
- [5] Blakemore, A., and Schebella, G. Tools for analyzing dynamic properties of system and software designs. In *Proceedings of the XI IFIP Congress, San Francisco, CA*. Elsevier Science Publishers B.V. (North-Holland), Aug. 1989, pp. 985-988.
- [6] Chiola, G. A software package for the analysis of Generalized Stochastic Petri Net models. In *Proceedings of the International Workshop on Timed Petri Nets*. Torino, Italy, July 1985, pp. 136-143.
- [7] Ciardo, G. Analysis of large stochastic Petri net models. PhD thesis, Duke University, Durham, NC, USA, 1989.
- [8] Ciardo, G., Trivedi, K. S., and Muppala, J. SPNP: stochastic Petri net package. In *Proceedings of the Third International Workshop on Petri Nets and Performance Models (PNPM89)*. Los Alamitos, CA, USA, Dec. 1989, pp. 142-151.

- [9] Csenki, A. Recovery block reliability analysis with failure clustering. In *Proc. of IFIP Working Group 10.4 Int. Working Conf. on Dependable Computing for Critical Applications*. University of California, Santa Barbara, Aug. 1989.
- [10] Duda, A. Approximate performance analysis of parallel systems. In Iazeolla, G., Courtois, P. J., and Boxma, O. J. (Eds.), *Computer Performance and Reliability*. Elsevier Science Publishers, B. V. (North-Holland), Amsterdam, The Netherlands, 1988, pp. 189-202.
- [11] Heidelberger, P., and Trivedi, K. S. Queueing network models for parallel processing with asynchronous tasks. *IEEE Transactions on Computers*. C-31, 11 (Nov. 1982), 1099-1109.
- [12] Heidelberger, P., and Trivedi, K. S. Analytic queueing models for programs with internal concurrency. *IEEE Transactions on Computers*. C-32, 1 (Jan. 1983), 73-82.
- [13] Hsueh, M. C., Iyer, R. K., and Trivedi, K. S. Performability modeling based on real data: a case study. *IEEE Transactions on Computers*. C-37, 4 (Apr. 1988), 478-484.
- [14] Laprie, J. C. Dependability evaluation of software systems. *IEEE Trans. Softw. Eng.*. SE-10, 6 (Nov. 1984), 701-714.
- [15] Leu, S.-W., Fernandez, E. B., and Khoshgoftaar, T. Fault-tolerant software reliability modeling using Petri net. *Microelectronics and Reliability*. 31, 4 (1991), 645-667.
- [16] Mak, V. W., and Lundstrom, S. F. Predicting performance of parallel computations. *IEEE Transactions on Parallel and Distributed Systems*. 1, 3 (July 1990), 257-270.
- [17] Musa, J. D., Iannino, A., and Okumoto, K. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, Singapore, 1987.
- [18] Peng, D., and Shin, K. G. Modeling of concurrent task execution in a distributed system for real-time control. *IEEE Transactions on Computers*. C-36, 4 (Apr. 1987), 500-516.
- [19] Peterson, J. L. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1981.

- [20] Petri, C. Kommunikation mit Automaten. PhD thesis, University of Bonn, Bonn, West Germany, 1962.
- [21] Pucci, G. On the modelling and testing of recovery block structures. In *Proc. Twentieth Int. Symp. on Fault Tolerant Computing*. Newcastle upon Tyne, UK, 1990, pp. 356-363.
- [22] Randell, B. System structure for software fault tolerance. *IEEE Trans. Softw. Eng.* SE-1, 2 (June 1975), 220-232.
- [23] Sanders, W. H., and Meyer, J. F. METASAN: a performability evaluation tool based on Stochastic Activity Networks. In *Proceedings of the ACM-IEEE Comp. Soc. Fall Joint Comp. Conf.* Nov. 1986, pp. 807-816.
- [24] Scott, R. K., Gault, J. W., and McAllister, D. F. The consensus recovery block. In *Proc. Total System Reliability Symp.* 1983, pp. 74-85.
- [25] Scott, R. K., Gault, J. W., and McAllister, D. F. Fault-tolerant software reliability modeling. *IEEE Trans. Softw. Eng.* SE-13, 5 (May 1987), 582-592.
- [26] Stansifer, R., and Marinescu, D. Petri net models of concurrent Ada programs. *Microelectronics and Reliability*. 31, 4 (1991), 577-594.
- [27] Stone, H. S. *High-Performance Computer Architecture*. Addison-Wesley, Reading, Massachusetts, 1987.
- [28] Thomasian, A., and Bay, P. Analytic queueing network models for parallel processing of task systems. *IEEE Transactions on Computers*. C-35, (Dec. 1986), 1045-1054.
- [29] Trivedi, K. S. *Probability & Statistics with Reliability, Queueing, and Computer Science Applications*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1982.
- [30] Woodside, C. M. Throughput calculation for basic Stochastic Rendezvous Networks. *Performance Evaluation*. 9, (1988/89), 143-160.

- [31] Woodside, C. M., et al. An active-server model for the performance of parallel programs written using rendezvous. *The Journal of Systems and Software*. 6, 1 & 2 (May 1986), 125-131.

The Network Synthesis System

presented to the
1992 Complex Systems Engineering
Synthesis and Assessment
Technology Workshop
at the
Naval Surface Warfare Center
White Oak, Silver Springs, Maryland

prepared by
Erwin H. Warshawsky
JRS Research Laboratories Inc.
Orange, California

The Network Synthesis System

Background

Modern weapon systems have extremely complex control, computational, and information processing requirements. When implemented, they include tens, or even hundreds, of computers tied together in networks with various degrees of intimacy. The DoD is faced with the problem of declining budgets, and the attendant reduction in both numbers of designers and numbers of engineering organizations, at the same time that mission complexity and system complexity are increasing greatly. This implies that the means must be found to greatly improve the productivity of the smaller cadre of designers and to create increasingly more complex and high quality systems with limited budgets.

The Network Synthesis System is an integrated design automation system for performing Network Synthesis for specific applications. The primary objective of the Network Synthesis System is to provide a smart designer with the tools that will enable him to mostly automatically synthesize a network. The System would provide facilities for synthesizing and evaluating numerous alternatives. It would enable the automated synthesis of network solutions based totally on the requirements (e.g., algorithm X must execute in 90 msec) and constraints (e.g., the resulting hardware must weigh no more than 5 lbs.) imposed by the designer. By deriving solutions from requirements and constraints, the System will ensure that solutions meet specifications and enable the quantification of the impact of specification changes. Other objectives of the Network Synthesis System are involved with such issues as Design Optimization Strategies, Validated Hardware and Software Parts Libraries, and User Friendly Interfaces.

A successful Network Synthesis System would be expected to reduce design time by a factor of 10 to 100. It would enable the practical development of proof-of-concept and pre-production prototypes in an almost wholly automated way. It would reduce the risk and cost in major system development. One could synthesize and evaluate a complex network in days with such a tool; a process that now takes months and years.

The NSS will be developed by integrating together two existing tool sets and by providing additional tools to support the network design process.

The two existing tool sets are:

1. Processing Graph Methodology (PGM)

The PGM is a tool set developed by the Naval Research Laboratory on the AN/UYS- 2 Program. PGM tools enable the design of an application system to be done at a high level, expressed in graph notation, and then be translated into code strings that are executable in the functional elements of the system. It includes a set of network level simulation tools, called PGSE, which provide designers with the means for evaluating the performance of their designs. It also includes comprehensive graph building tools.

2. Integrated Design Automation System (IDAS)

IDAS is a tool set developed by JRS mostly under the sponsorship of the Tri- Services VHSIC Program. The JRS tools include an automatically retargetable Ada Compiler, an Ada Behavioral Specification to VHDL Structural Description Synthesizer, and various tools to assist in adapting Ada to the special characteristics of an arbitrary embedded computer and supporting application programmers in achieving effective results. The JRS tools are unique, particularly as they relate to Ada and VHDL. Various Simulators are also included.

Some of the new tools to be developed include:

- User Interface to control the design process
- Strategies and algorithms for the various design optimizations.
- Library Building and Management Facilities for collecting, storing, and accessing relevant data.

System Summary

The Network Synthesis System (NSS) will provide a Designer with the tools and methodology needed to synthesize network level systems consisting of computers of various types and capabilities (e.g., signal processors), memories of various types and sizes, input/output elements of various types (e.g., sensors, displays, controls), and the communication elements of various types needed to link the pieces of a network together in an effective manner. With the NSS, a Designer will be able to rapidly prototype (in model form) a network and evaluate it objectively against its specifications (e.g., performance, functionality) and constraints (e.g., limits on power, size, reliability).

The NSS will provide an integrated collection of tools to support a comprehensive design methodology. It will include:

- Network Synthesis Tools
- Processor Synthesis Tools
- Concurrent Hardware/Software Design
- Ada and VHDL Languages
- Specification First Design
- Reusable Part Libraries for both Software (Ada) and Hardware (VHDL)
- Simulation and Evaluation Tools covering the Design Hierarchy from Networks to Components
- Estimating Tools incorporating Rules of Thumb and Engineering Judgment
- Physical Package Modelling, Partitioning, and Assignment
- Integration with Lower Level Tools (e.g., Silicon Compilers)

Network Synthesis from Specifications and Constraints

The network synthesis process is driven by the network requirement specifications. This means that the NSS will take, as input, a behavioral specification of the network to be synthesized and will be capable of automatically generating, as output, a structural description of a network that satisfies the behavioral specification.

The network synthesis process is controlled by the Designer via the imposition of constraints on the solution space available to the NSS in transforming the behavioral specifications into structural descriptions. "Constraints" are to be viewed as budgets or limits. The network of interest may, for example, be budgeted to no more than 100 watts or a 2000 hour MTBF; it may be limited to three computers or to only computers for which validated Ada compilers exist. These "constraints" will directly affect the solution that is generated.

Each particular set of behavioral specifications and constraints imposed on the solution will, in general, lead to a different solution. The comparison of the different solutions, in terms of any parameters of interest, is called a tradeoff. Designers perform "tradeoffs" to measure the sensitivity of some solution parameters to changes in one or more of the behavioral specifications and constraints. They perform "what if?" experiments. Designers will vary the specifications of the required behavior (e.g., change the image processing algorithm) or modify one or more constraints (e.g., up the Power budget to 200 watts), then regenerate the solution, and then compare the solution parameters to those of other, previously generated, solutions. Designers are generally looking to find the best solution, that satisfies all of the requirements; a 50 watt solution is better than a 100 watt solution, even if the budget is 200 watts.

Behavioral specifications or constraints need not be complete or finished in some sense. The NSS will generate solutions based on the inputs presented and, in fact, will not know, or be concerned with, whether or not the inputs are correct or complete. This capability facilitates what if experiments. It is necessary for a methodology wherein a design evolves as more knowledge of the requirements is gained or a design must respond to an abrupt change in specifications or constraints. It is also necessary for a methodology that supports the concept of providing reasonable estimates of the implications of design decisions; that is, the System will be able to objectively estimate the difference in the power requirement, for example, of a design implemented in GaAs versus CMOS.

The solution to a network synthesis problem is a structural network. The structural network consists of a collection of hardware nodes, of a variety of types (e.g., computers, sensors), interconnected with one another in some manner. A network solution is obtained when all of the nodes and all of the interconnections are realized; realized means that hardware has been selected or designed to implement the node or interconnection; the solution obtained must satisfy the network specifications and constraints.

Processor Synthesis from Specifications and Constraints

In synthesizing a network, a solution will be sought that utilizes existing hardware; that is, hardware that is modelled and contained in a reusable parts library and for which a physical implementation

may exist.

At times, it will be found that the requirements are such that no solution can be found when restricted to the use of existing hardware. Then, the requirement to synthesize a new part (e.g., a computer) might arise. In a similar vein, it may simply be of interest to explore the design space and generate new library entries for later network problems. In either case, whether the part synthesis requirement is generated by a network synthesis problem or by an off-line library building/validating process, a part level synthesis tool is needed to satisfy the requirement.

The NSS provides a Processor Synthesis tool set that is driven by behavioral specifications and constraints. Behavioral specifications are expressed as Ada Programs. Processor synthesis results in a solution, which is a computer description expressed in VHDL; Ada specifications to VHDL Descriptions.

Processors are synthesized from parts in a library, exactly as is done for network synthesis; the parts, at this design level, are things like ALUs, Memories, Registers, etc.

Processor synthesis includes numerous optimization elements for effecting the generation of good quality designs, not simply functionally correct ones.

Concurrent Hardware/Software Design

The NSS supports concurrent hardware/software design. It provides three main capabilities in this regard:

- Provides facilities for performing comprehensive hardware/software tradeoffs and for measuring the impacts of decisions objectively and quantitatively.
- Provides an automatically retargetable Ada Compiler System, that is retargeted from a description of a computer expressed in VHDL.
- Provides comprehensive analysis and tracing facilities, coupled with a sophisticated User interface, to allow Users to see the detailed relationship between the hardware and software. For example, one can see detailed hardware utilization data associated with the execution of any selected program; this would be of great interest for real time evaluations and for fault tolerance analyses.

Together, these capabilities allow a Designer to proceed with hardware and software design and implementation concurrently. It allows for evaluation of designs almost instantaneously (i.e., within hours) from the time that versions are identified. It allows detailed development to start early and proceed with confidence. Integration and testing is started immediately, not at the end of the project.

Standards – Ada and VHDL

The NSS utilizes Standards where ever practical in its design and development. It is anchored to the use of Ada and VHDL.

Ada is utilized within the NSS in the conventional manner as an application programming language. It will be used as the application programming language for all types of applications and for all types of computers. Ada is also used as the behavioral specification language for a computer in a network; normal, compilable and executable, Ada programs will be used for this purpose.

VHDL is used in the NSS as the language for describing hardware, at any level in the system hierarchy from network to gate. The NSS will contain and/or generate VHDL models of networks, computers, and lower level components.

Specification First Design

From a language point of view, synthesis at any design level is a process that ultimately transforms a behavioral specification into a structural description. This implies that one should first determine what an entity is to do (i.e., its behavioral specification) and then generate and examine alternative designs (i.e., solutions) that do it; this is an eminently reasonable idea. The NSS supports this methodology very strongly.

The NSS uses Graphical Representations and Ada Programs as the primary mechanisms for expressing behavioral specifications.

At the network level, the NSS utilizes data flow graphs to depict the behavior and to express the concurrency possible in the behavior. Nodes in a graph are reducible to a "primitive" level at which the behavior of the node is expressible as an Ada Program. The network synthesis process transforms an arbitrary behavioral network, containing an arbitrary number of primitive nodes, into a structural network, having a specified number of hardware nodes, of specified types, interconnected in a specified manner. Normally, a network will exhibit concurrency of behavior on a "primitive" node level.

At the processor level, the NSS utilizes Ada Programs to depict the behavior required and to express the concurrency possible in the behavior (as discernible in the data dependency graphs for the program). The Ada Programs used may have originated as primitive node specifications on the network level; if so, the connection from network level to processor level is seamless.

The primitive behaviors at the processor level are expressed as arithmetic or logical operations. These behaviors are expressible as logic equations, truth tables, or similar representations, which can then be passed to logic synthesizers for synthesis of elements like register files and RALUs.

Reusable Parts – Hardware and Software

The NSS incorporates, and relies heavily on, the concept of reusable parts. It includes libraries of hardware elements (e.g., processors, displays, ALUs, memories) and software elements (i.e., algorithms

expressed in Ada). It also includes "reusable data" elements that represent the relationships between parts in reusable libraries; for example, reusable data includes the execution time of an algorithm (from the reusable Ada library) on a processor (from the reusable computer library in VHDL) as a data element that is maintained.

Synthesis on a network level involves reusable parts very heavily. Primitive behavioral nodes are from the Ada library. Selectable computers are obtained from the VHDL library when constructing the structural graph. Relational data elements are used in deciding how to assign behavioral nodes to structural elements.

Reusable Part Libraries are conceptually very important for two other reasons:

- Provides the mechanism for enforcing "validation" of parts and the use of validated parts
- Facilitates the synthesizing of parts, for all design levels, in a manner independent of the technical and schedule requirements of a project. Algorithm designers, for example, could do their thing and deposit the results into a library off-line from any specific project

Simulation and Evaluation Tools

The NSS contains the capability to simulate and evaluate designs from the network to the component level. It contains the following:

- Network Level Behavioral Simulation
- Network Level Structural Simulation
- Processor Level Software Simulation (VHDL)
- Processor RTL Level Simulation (VHDL)
- Component Level Simulation (VHDL)

These tools are tightly integrated into the design methodology and are used to generate data upon which design selection and optimization decisions are made.

Estimating Tools

The NSS includes the capability of rapidly generating estimates. When performing synthesis from a high level, network or processor, a designer will be interested in estimates of various design parameter values that would result from alternative constraints that might be imposed. For example, the designer would like to know the estimated MTBF for a given design implemented in GaAs and Packaging Scheme 1 versus CMOS and Packaging Scheme 3.

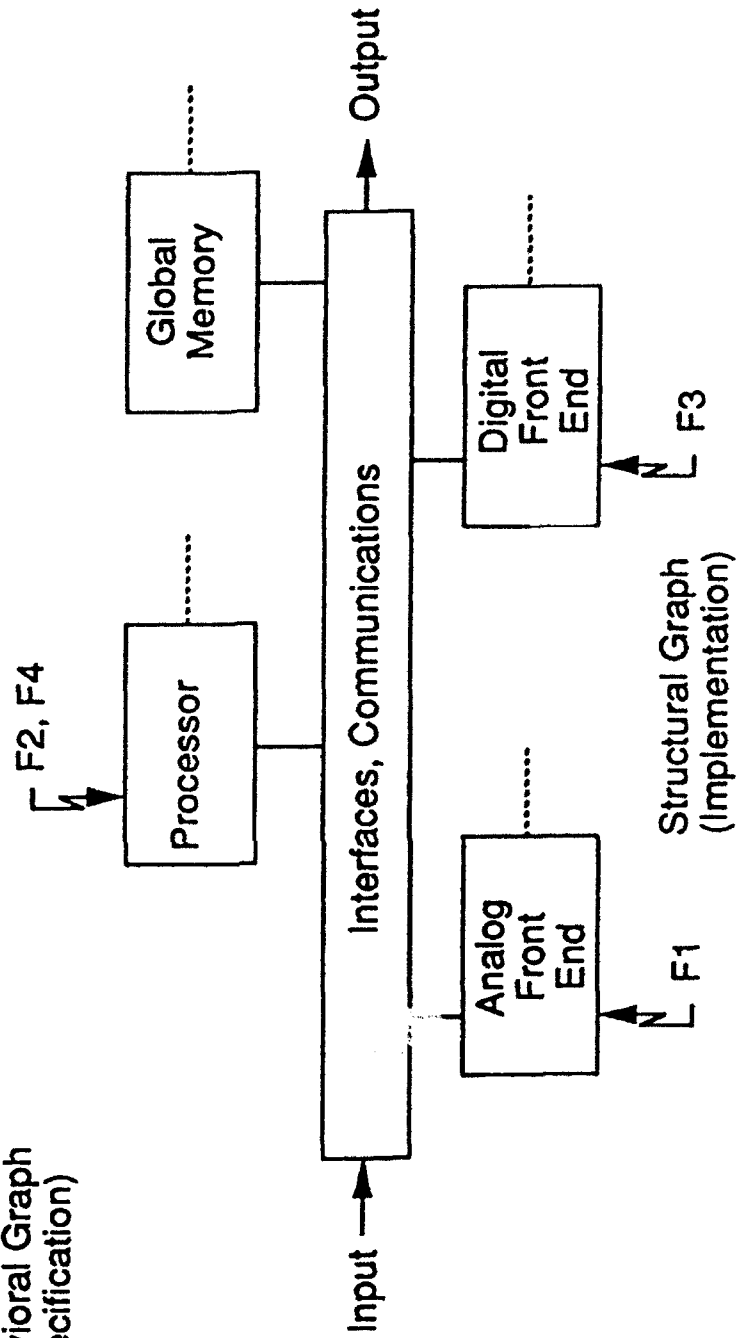
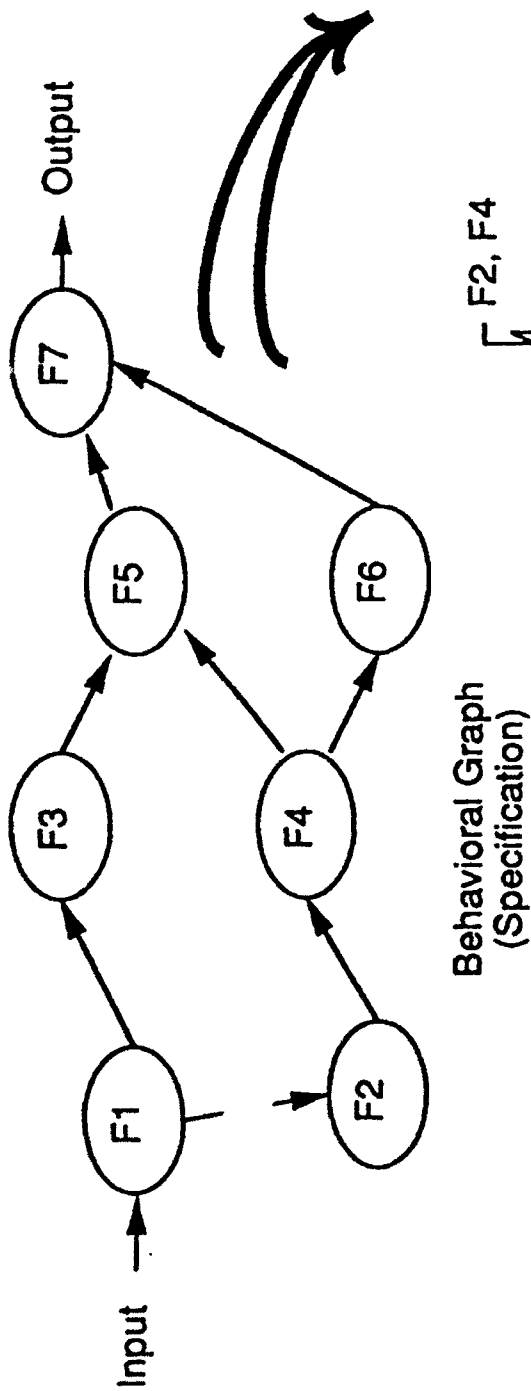
The NSS provides rapid estimate generation based on measures of complexity and "good engineering judgment" or "rules of thumb". The capability is data driven, which means that any organization or individual may establish unique estimating factors for any combination of constraints of interest.

Physical Package Modelling, Partitioning, and Assignment

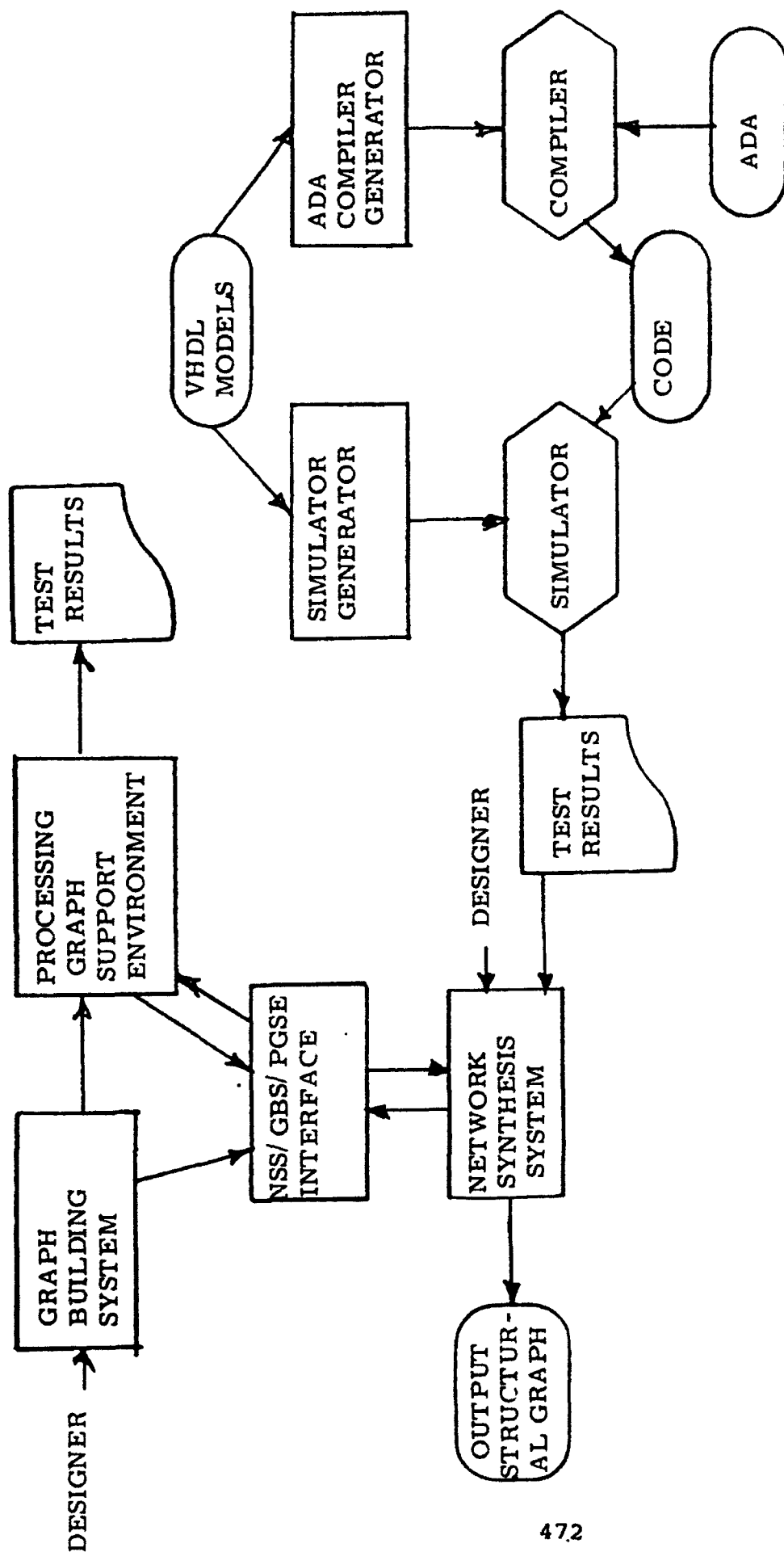
The NSS provides the facilities for modelling a set of physical packages (e.g., chassis, board, carrier, hybrid, IC) in terms of their capacities. It also provides the capability of partitioning an interconnected set of hardware components, at any structural level, into subsets that match the capacities of the packages. Together, these capabilities provide the ability to do multilevel assignment.

Integration with Lower Level Tools

The NSS utilizes standard languages and formats to facilitate its interconnection to other tools. It has been interfaced to VHDL compiler systems, a silicon compiler system, various simulation tools, and various software assemblers/linkers/loaders. It is soon to be interfaced to a MOSIS supported silicon compiler.



Network Synthesis



Reliability of Redundant Arrays of Inexpensive Disks (RAID) *

Manish Malhotra
Department of Computer Science
Duke University
Durham, NC 27706
malhotra@cs.duke.edu

Kishor S. Trivedi
Department of Electrical Engg.
Duke University
Durham, NC 27706
kst@egr.duke.edu

June 5, 1992

Abstract

Reliability analysis of various disk array architectures (different levels of RAID) is performed. The dependence of reliability and mean time to data loss on different parameters of a disk array and support hardware components needed for correct functioning of disk array is characterized. A study of these characteristics reveals the implications of several design issues of a disk array on its reliability. Issues like scalability of disk arrays, imperfect coverage of disk failures, cold versus hot disk spares, predictive disk failures, reliability of disk arrays for mission-critical computer systems, serial versus orthogonal placement of support hardware with respect to disk groups, and levels of hardware redundancy are studied.

1 Introduction

To achieve high computer systems performance, the performance of its components must increase in proportion to each other. Unfortunately, I/O storage systems have not been able to match the high performance of the CPUs and memories. To bridge this performance gap, high-performance disk array architectures have been proposed. Given similar performance and cost per megabyte, higher performance can be obtained by using an array of smaller disks compared to a single large disk. More arms can be provided and requests that access only a single disk can be serviced independently. Large requests that need to access several disks can be serviced much faster by performing data-transfer in parallel.

*This work was supported in part by the National Science Foundation under Grant CCR-9108114 and by the Office of Naval Research under Grant N00014-91-J-4162

However, an array of disks is more fault-prone than a single large drive. Assuming the MTTF (mean time to failure) of a large disk drive to be the same as the MTTF of a small disk, an array of a hundred disks would have an MTTF that is one hundredth of that of a large drive. Even if the assumption made above is not correct, it is not hard to see that the reliability of a disk array system can be much less than required. Failure of a disk may result in loss or corruption of data. In many applications, data is accumulated through years of research efforts and extensive experimentation. Loss of critical data is clearly unacceptable as it may lead to financial loss or loss of life. Thus, the demand is to design cost-effective disk systems which can not only deliver high-performance but also provide high reliability.

The purpose of this paper is to quantify the reliability and mean time to data loss of disk array architectures and provide answers to important questions arising in the design of a disk array by means of analytic models. Gibson [3], Schulze et al [15], and Patterson et al [12] have analyzed reliability of different RAID architectures in terms of mean time to data loss (MTDL). Bitton and Gray [1] have analyzed MTTF for mirrored disks. However, these approaches use simple approximations. The most comprehensive effort to analyze RAID reliability in terms of different parameters is the work by Gibson and Patterson [4]. They compute MTDL for disk array models based on certain approximations. For these models, they assume that time to failure of a *group* of disks is exponentially distributed. Based on this assumption, they compute the reliability of disk arrays using the approximate value of MTDL. Usefulness of these approximations is that MTDL and reliability can be expressed in closed form.

It is easy to see that time to failure of a group of disks is not exponentially distributed even when individual disk failure times are. Moreover, the emphasis on MTDL as a metric to evaluate the reliability of disk arrays could be misleading. The variance of time to data loss for a typical disk array is very large and the actual time to data loss in practice may differ significantly from the MTDL. In this paper, we develop hierarchical reliability models for RAID architectures and perform exact analysis of these models. Use of hierarchical modeling provides a better understanding of the architecture being modeled and keeps the state space of models small. To make the models realistic, we take into account several

factors which have hitherto not been considered, mainly, imperfect coverage of disk failures, disk failure prediction, and the type of spares (cold or hot). Even for slightly advanced models, closed form solutions become very messy or impossible. Therefore we provide numerical solutions. We emphasize reliability more than the MTDL as a metric to evaluate different RAID architectures.

Our analysis not only provides a comparison between different disk array architectures, but also provides answers to specific questions about disk arrays in general, such as : 1) How reliable should each individual disk be? 2) Are disk arrays reliable enough for mission-critical systems? 3) Should the disk spares be kept hot or cold? 4) How much redundancy in disk spares is needed? 5) Just how small should the data reconstruction time be? 6) Should hardware redundancy be scaled as the dimensions of disk arrays are scaled? 7) How much better is the orthogonal placement of support hardware than serial placement? and

The rest of this paper is organized as follows. In Section 2, we describe a general hierarchical reliability model for a general disk array architecture. This model fits several of the disk array architectures which we briefly describe. In Section 3, we extend this model for the architectures which rely on disk controllers for disk failure detection and location. In Section 4, we develop reliability models for different RAID architectures with finite number of cold and hot disk spares. In Section 5, we include support hardware components into our model. We develop models for two different hardware organizations for RAID : serial and orthogonal placement of support hardware with respect to disk groups. In Section 6, numerical results obtained from the solution of these models are presented and discussed. Finally, we present our conclusions based on these results in Section 7.

2 Fault-Tolerant Disk Array Architectures

Several fault-tolerant disk array architectures have been introduced by different researchers using varying degrees of hardware redundancy [1, 7, 8, 9, 11, 14]. Patterson et al [11] coined the term *RAID (Redundant Array of Inexpensive Disks)* for such disk array systems with redundancy. They unified the existing disk system architectures as different levels of RAID (levels 1,2,3,4) and proposed a new high-performance disk system architecture (RAID level

5).

We now introduce the terminology and state the assumptions used throughout the paper. The disk array consists of N groups of disks. Each group has D data disks and C check disks. We assume that time to failure of each disk is exponentially distributed with mean $1/\lambda$ (MTTF). Failure of a single disk in a group is tolerable since lost data can be reconstructed. Data reconstruction process consists of two steps : disk replacement and data construction on the replaced disk. Disk replacement consists of bringing in a spare disk to replace the failed disk. Data construction is accomplished using the data from the working disks and parity information of the group. We initially assume that each group has a spare disk which can be electronically switched in when a disk fails. We further assume that the spare disk does not fail as long as it is not switched in.

However, if another disk in the same group fails while the reconstruction is underway, then data is lost (can not be reconstructed) and that group of disks is considered failed. We assume that each group has its own reconstruction mechanism independent of other disks. Thus, reconstruction could be carried out for more than one group at the same time. All the disks are identical (come from the same manufacturer). Unless otherwise stated, the data reconstruction time is assumed to be exponentially distributed with mean $1/\mu$. Failure of any group results in the failure of disk array so that data loss in any group constitutes the failure of disk array.

We define *data-reliability* as the probability that no data loss occurs until time t . Thus, data-reliability is same as the reliability of disk array and it is expressed as a function of time t . Based on these assumptions, we construct a two-level hierarchical reliability model for a general disk array architecture as described in [6]. The reliability of the disk array is modeled by a reliability block diagram (RBD) shown in Figure 1. This upper level model has a series structure. Each block represents a group of disks. Until Section 5, we assume that groups behave independently of each other. If $R_i(t)$ is the reliability of group i , then reliability of the disk array is given by :

$$R_{da}(t) = \prod_{i=1}^N R_i(t) . \quad (1)$$

To compute the reliability of a group, we use a simple Markov model shown in Figure 2.



Figure 1: Reliability block diagram for RAID

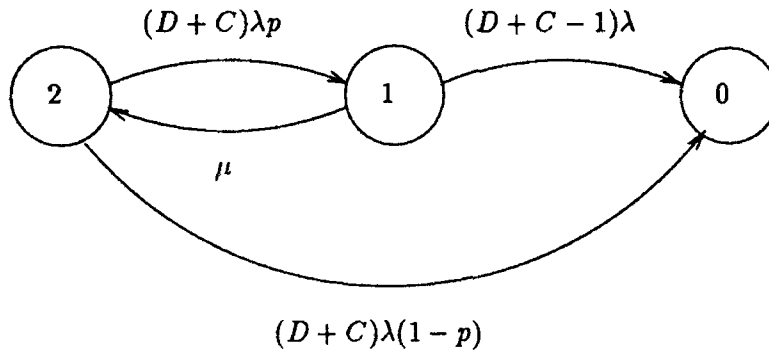


Figure 2: Markov reliability model for a single group of disks (RAID-1,2,3,4,5)

In state 2, all the disks in a group are operational. After one of the disks fails, system state changes from 2 to 1 and data reconstruction is initiated. However, the disk array keeps functioning since data is available. If any other disk in the group fails before the reconstruction is completed, then data is lost and the disk array is considered failed. State 0 is the group failed state.

Note that we allow imperfect coverage of faults. A fault in a disk is covered with probability p and not covered with probability $1 - p$. An uncovered fault in a group causes data loss. Bit errors that are not detected, not corrected, or miscorrected are manifestations of what we call uncovered faults. These faults may occur due to some fault within the error-correcting code (ECC) or if ECC is not properly invoked when an error is detected. Moreover, ECC can correct only single bit errors. Occurrence of multiple bit errors (which may happen due to some extraneous electric signal), are accounted for by imperfect coverage. Failures in support hardware (e.g., failure of cooling equipment) may cause an unrecoverable failure in a disk. Some disk array architectures rely on the array controller's ability to

detect disk failures. Failure of the disk controller mechanism to detect a disk failure is also considered an uncovered failure. Imperfect coverage also accounts for catastrophic failures due to extreme environmental conditions.

The reliability of any group G_i is given by :

$$R_i(t) = A_1 e^{\beta_1 t} + A_2 e^{\beta_2 t}, \quad (2)$$

where

$$\begin{aligned} \beta_1 &= \frac{-(\mu + (2D + 2C - 1)\lambda) + \sqrt{\lambda^2 + \mu^2 + \lambda\mu((4D + 4C)p - 2)}}{2}, \\ \beta_2 &= \frac{-(\mu + (2D + 2C - 1)\lambda) - \sqrt{\lambda^2 + \mu^2 + \lambda\mu((4D + 4C)p - 2)}}{2}, \\ A_1 &= \frac{((D + C)(1 + p) - 1)\lambda + \mu + \beta_1}{\beta_1 - \beta_2}, \\ A_2 &= \frac{((D + C)(1 + p) - 1)\lambda + \mu + \beta_2}{\beta_2 - \beta_1}. \end{aligned} \quad (3)$$

The reliability of the disk array is given by Equation 1. Mean time to data loss for a group of disks is :

$$MTDL_i = \frac{\mu + ((D + C)(1 + p) - 1)\lambda}{(D + C)\lambda(\mu(1 - p) + (D + C - 1)\lambda)}. \quad (4)$$

The mean time to data loss for the disk array is given by :

$$MTDL_{da} = \int_0^\infty R_{da}(t)dt = \sum_{j=0}^N \frac{\binom{N}{j} A_1^j A_2^{N-j}}{\beta_1 j + \beta_2 (N - j)}. \quad (5)$$

2.1 RAID-1 (Mirrored Disks)

Mirroring is the traditional approach to improve the reliability of disk systems. Bitton and Gray [1] introduced the concept of disk shadowing in which a shadow set of k disks (i.e., k identical copies of same data) are maintained. This set can support k reads in parallel assuming parallel data paths and enough disk controllers (thus effectively increasing the read rate by a factor of k). A write is performed in parallel over k disks (thereby maintaining a write rate of single disk). We concern ourselves with the case where $k = 2$. This configuration is also known as disk duplexing or mirroring.

If the storage capacity of the system requires N disks, then $2N$ disks are used in a duplex system. This is the most expensive of the different RAID architectures. It is also

significantly different from the rest of RAID architectures. Each pair of mirrored disks forms a group. If one of the disks in a group fails, then a spare is switched in. The data reconstruction consists of copying the data onto the spare from the other working disk. For the case of RAID-1, we substitute $D = 1$ and $C = 1$ in Equations 1 - 5.

2.2 RAID-2

In this scheme, each group has D data disks and C (where $C \geq \log_2(D + C + 1)$) check disks [5]. The check disks can correct single bit errors and detect double bit errors. A single failure of any disk in a group is tolerable. An uncovered fault or failure of two or more disks causes data loss. Substituting the appropriate values of D and C in Equations 1 - 5 yield the reliability and MTDL for this organization for RAID.

2.3 RAID-3,4,5

The C check disks for D data disks in RAID-2 are basically needed to detect the incorrect bit position. Once the incorrect bit has been identified, then a single parity bit suffices for correction (reconstruction) of data which would otherwise be permanently lost. In RAID levels 3,4, and 5, the ability of the disk controller to detect a failed disk is utilized. Thus, we need only one check disk per group since the disk controller identifies the failed bit position.

A RAID-3 architecture was proposed by Park and Balasubramaniam [10] and a RAID-4 architecture was proposed in [14]. RAID-4 differs from RAID-3 in that the data is interleaved between disks at sector level. In RAID-3, data is interleaved at bit level. Thus, in RAID-4, an I/O transfer is spread across all the disks within a group. Whereas RAID-3 allows only one I/O transfer per unit time per group, RAID-4 allows parallel transfers from a group. However, only the reads are parallelized. Writes are limited to one per group at a time since every write request results in a read and write to the parity disk. Therefore, RAID-4 results in improved performance for reads.

To parallelize writes, RAID-5 architecture was proposed by Patterson et al [11]. In this scheme, parity information is spread across all the disks within a group (rotated parity). This scheme results in improved performance for reads as well as writes. However, the reliability models for RAID-3, RAID-4, and RAID-5 are identical since they all require only

one check disk. Substituting $C = 1$ in Equations 1 - 5 yields the desired results for reliability and MTDL.

3 Predictive Disk Failures

RAID-3,4, and 5 rely on the disk controller's ability to correctly predict the disk failures before they occur. Some implementations utilize this property to prevent data loss and reduce the reconstruction time. We modify our earlier model to account for these new features. We assume that no loss of data occurs if the disk controller correctly predicts an impending disk failure. We further assume that the spare is electronically switched in and data copied onto the spare before the failing disk is powered down. This sequence of operations does not result in a change of state of the system. However, the disk controller may not always be able to predict a disk failure. Failures resulting from uncovered faults are not predictable. With probability $(1 - \alpha)$, an impending failure due to a covered fault is not predicted.

There is also the possibility of false alarms when the disk controller erroneously predicts a disk failure. The time to next false alarm is assumed to be exponentially distributed with rate γ . However, false alarms are treated as correctly predicted failures and do not result in a change in system state. This is because we assume unlimited supply of disk spares. However, it does result in monetary loss since a false alarm results in undue consumption of disk spares. In a later section, when we consider finite number of spares, this effect of false alarms is clear. The Markov reliability model based on these assumptions for each group is shown in Figure 3.

The reliability of each group has the same form as Equation 2 where

$$\begin{aligned}\beta_1 &= \frac{-(((D+C)(2-p\alpha)-1)\lambda + \mu) + \sqrt{X}}{2}, \\ \beta_2 &= \frac{-(((D+C)(2-p\alpha)-1)\lambda + \mu) - \sqrt{X}}{2}, \\ A_1 &= \frac{\beta_1 + \mu + (D+C-1)\lambda}{\beta_1 - \beta_2}, \\ A_2 &= \frac{\beta_2 + \mu + (D+C-1)\lambda}{\beta_2 - \beta_1},\end{aligned}$$

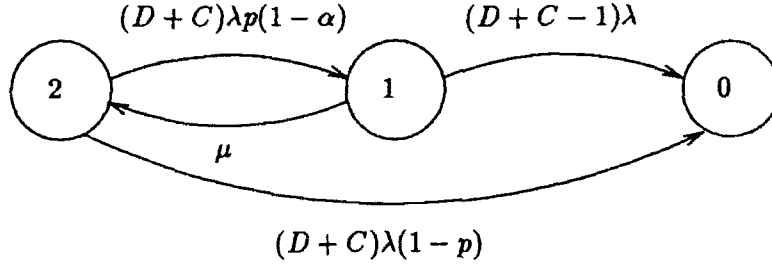


Figure 3: Markov reliability model of a group of disks with predictive failures (RAID-3,4,5)

$$X = ((D+C)((D+C)p^2\alpha^2 + -2p\alpha) + 1)\lambda^2 + \mu^2 + 2((D+C)p(2-\alpha) - 1)\lambda\mu.$$

Mean time to data loss for a group of disks is given by :

$$MTDL_i = \frac{\mu + ((D+C)(1+p(1-\alpha)) - 1)\lambda}{(D+C)\lambda(\mu(1-p) + (D+C-1)(1-\alpha)\lambda)} . \quad (6)$$

The reliability block diagram for the disk array is the same as shown in Figure 1. The reliability of the disk array is computed by Equation 1 and MTDL for the disk array is computed as in Equation 5.

4 Cold Disk Spares Versus Hot Disk Spares

In the earlier sections, we assumed unlimited supply of disk spares that did not fail. In reality, however, a fixed number of spares is maintained. The disk spares could be maintained *hot* or *cold*. A hot disk spare can fail even though it is not in active use. A cold disk spare does not fail unless it is switched in as a replacement for a failed disk. A hot spare can be switched in electronically after a disk fails and the time to perform the switch-in is negligible. Hence the effective data reconstruction time in this case consists only of constructing data on the spare disk. The disadvantages of hot disk spares are : 1) The automated switch-in mechanism adds to the cost overhead, 2) Spares can fail while not in active use, and 3) The hardware used to carry out spare switch-in may fail (these failures can be accounted for by the coverage probability).

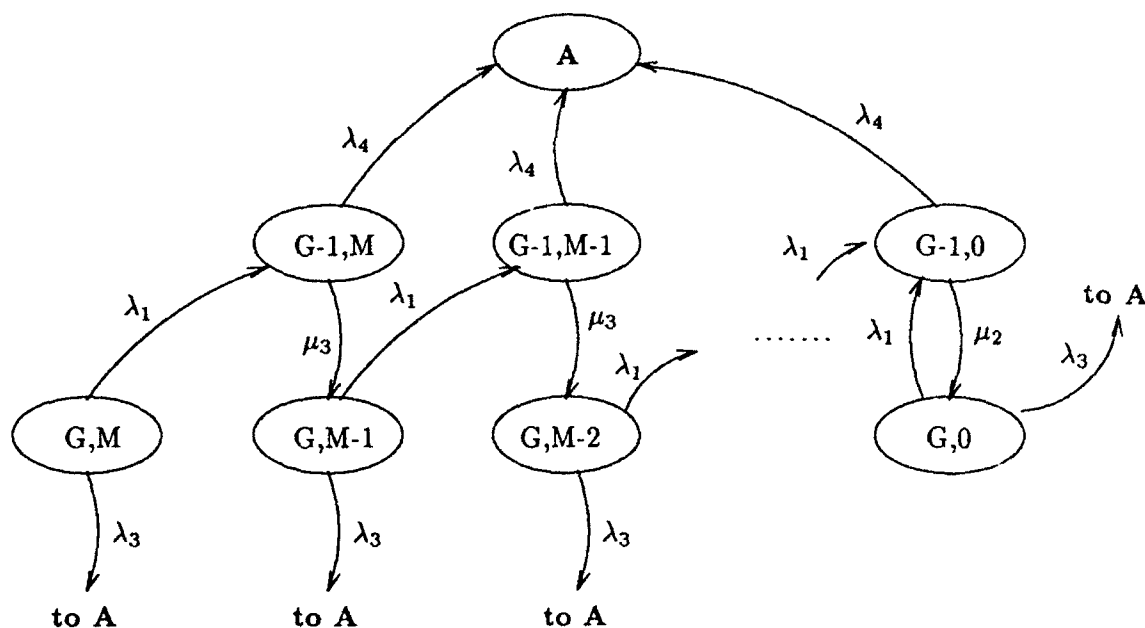


Figure 4: Reliability model of a group of disks with M cold spares (RAID-1,2)

The other option is to maintain cold disk spares. When a disk fails, typically a repair-person is called to install a spare. After installation, the disk reconfiguration and data reconstruction begins. Thus the total repair-time increases. A better solution perhaps is a combination of two approaches. Few hot spares could be maintained, while the rest are kept cold. Each time a disk fails, a hot spare is used up and a cold spare is made hot.

If a disk fails after all the spares are exhausted, then a new disk is ordered from the manufacturer. This increases the data reconstruction time. In practice, this could be avoided by always maintaining a minimum number of spares. Each time the number of available spares falls below this minimum, new disks can be ordered from the manufacturer. This scenario well approximates the case of unlimited spares. Maintaining spares (hot or cold) is a cost overhead. Some users may prefer not to maintain any spares. This strategy could result in savings depending upon several factors including loss of revenue while data reconstruction takes place and the reliability and availability desired. If disk spares are maintained, then the question arises as to how many spares should be kept? Intuitively, it

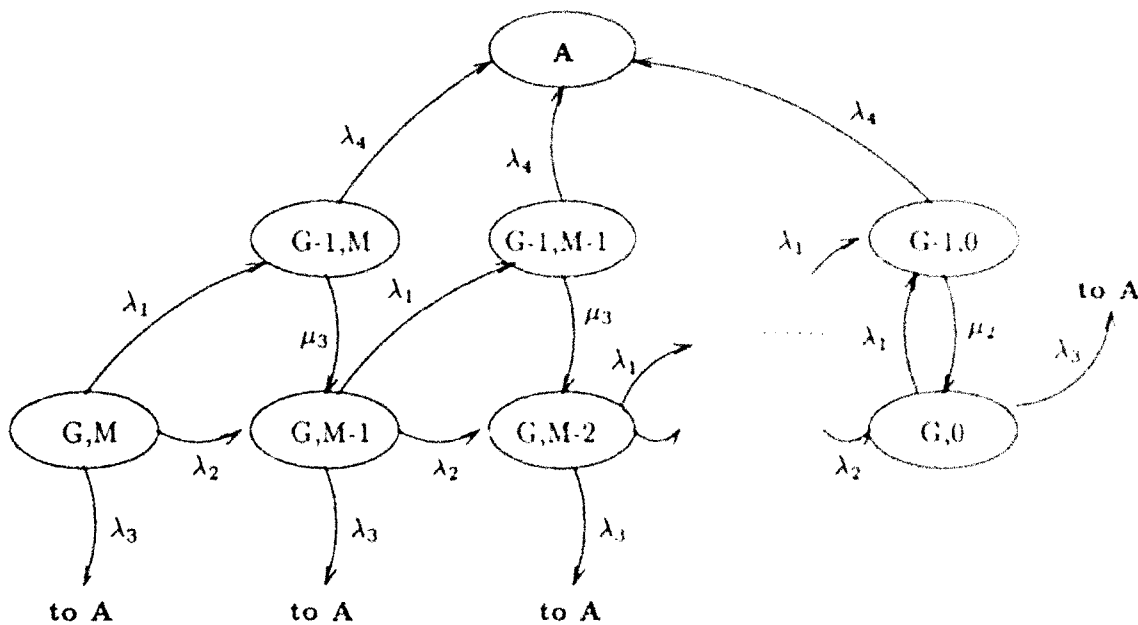


Figure 5: Reliability model of a group of disks with M cold spares and predictive failures (RAID-3,4,5)

is clear that if there are small number of groups each containing a large number of disks, then number of spares per group should be large. However, if there are a large number of groups each containing a small number of disks, then number of spares per group should be small.

In this Section, we develop reliability models for different RAID architectures based upon different assumptions. Assume that each group has M spare disks. Assume that for hot disk spares, the time to switch in a spare is negligible. Let us first consider the reliability model of a group of disks with M cold spares for RAID-1,2 (Figure 4). A state is a two-tuple (i, j) where i is the number of active disks (operational data and check disks) and j is the number of disk spares left. State A is the group failed state. In this model, $G = D + C$, the number of active disks, $\lambda_1 = (D + C)\lambda_d p$ where λ_d is the failure rate of a disk in active use, $\lambda_4 = (D + C - 1)\lambda_d$, and $\lambda_3 = (D + C)\lambda_d(1 - p)$. The transitions with rate λ_4 are transitions representing failure of a disk during data-reconstruction which results in

data loss. Transitions with rate λ_3 represent uncovered failure. Transitions with rate λ_1 represent covered failure of a disk. Rate of data-reconstruction is μ_3 as long as at least one disk spare is available. In state $(G,0)$, all the spares are exhausted. If a disk fails in this state, the data-reconstruction rate is μ_2 where $\mu_2 < \mu_3$ because mean data-reconstruction time increases.

In case of RAID-3,4,5 with cold spares, the reliability model is shown in Figure 5. Here $\lambda_2 = (D + C)\lambda_d p \alpha + \gamma$ and $\lambda_1 = (D + C)\lambda_d p (1 - \alpha)$ where α and γ are the same as defined in Section 3. Assume that data-reconstruction time for correctly predicted failures and false alarms is negligible. A spare is installed before the failing disk is powered down. Thus, these transitions result only in a state change reflecting the decrease in the number of available spares by one. The rates $\lambda_3, \lambda_4, \mu_3$, and μ_2 are same as the previous model.

Let us now consider reliability model for RAID-3,4,5 with hot spares (Figure 6). This model differs from the earlier model in that there are transitions from states $(G - 1, M - i + 1)$ to states $(G - 1, M - i)$ ($i = 1, M$) signifying the failure of hot disk spares. This also changes the transition rates from states $(G, M - i + 1)$ to $(G, M - i)$ (where $i = 1, M$). Rate $\lambda_{s,i} = i\lambda_{sp} + (D + C)\lambda_d p \alpha + \gamma$ where λ_{sp} is the failure rate of a hot disk spare and $\lambda_{sp} < \lambda_d$. Data-reconstruction rate while at least one spare is available is μ_1 and $\mu_2 < \mu_1$. In case of RAID-1,2 with hot spares, the reliability model remains the same but the transition rates change. Particularly, $\lambda_1 = (D + C)\lambda_d p$ and $\lambda_{s,i} = i\lambda_{sp}$.

5 Reliability Model of RAID with Support Hardware

A disk array system has many hardware components that are needed for proper functioning of the disk array. These include host bus adaptor (HBA), disk array controller (DC), hard disk drive (HDD) controller, single board controller (SBC) (track buffer and error correction circuitry (ECC) are resident in SBC), cooling hardware, and power supply etc. So far we have considered only the disks in the reliability models of disk arrays (coverage probability accounted for failure of some support hardware components though). Schulze et al [15] have shown that failures of the support hardware considerably reduces the overall reliability of a disk array. In fact, failure of some of the support hardware components may result in

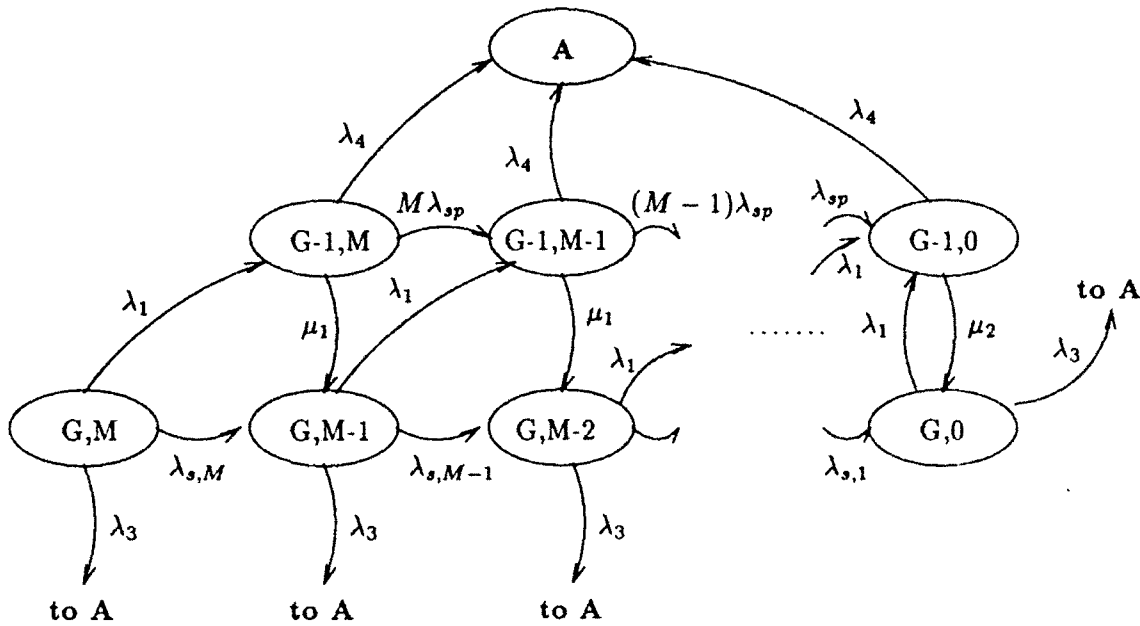


Figure 6: Reliability model of a group of disks with M hot spares

data loss. For instance, failure of cooling equipment may result in data corruption in disks and failure of power supply may result in data loss. In this Section, we model failure-repair behavior of support hardware components on reliability of disk array. We consider two hardware organizational schemes.

5.1 Serial Placement of Support Hardware

This is a simple organizational scheme in which the disk array has a set of associated hardware components (HBA, power supply (PS), cooling fans (CF), HDD, SBC, disk controller etc.). These components are placed serially with the disk array. If there is no redundancy in these components, then failure of any of these components results in the failure of the disk array. If there is redundancy, then failures of some of these components can be tolerated and failed components may be repaired or replaced. However, the placement is serial and if any component were to stop functioning despite the redundancy, the disk array is considered failed. The two-stage hierarchical reliability model is shown in Figure 7. The

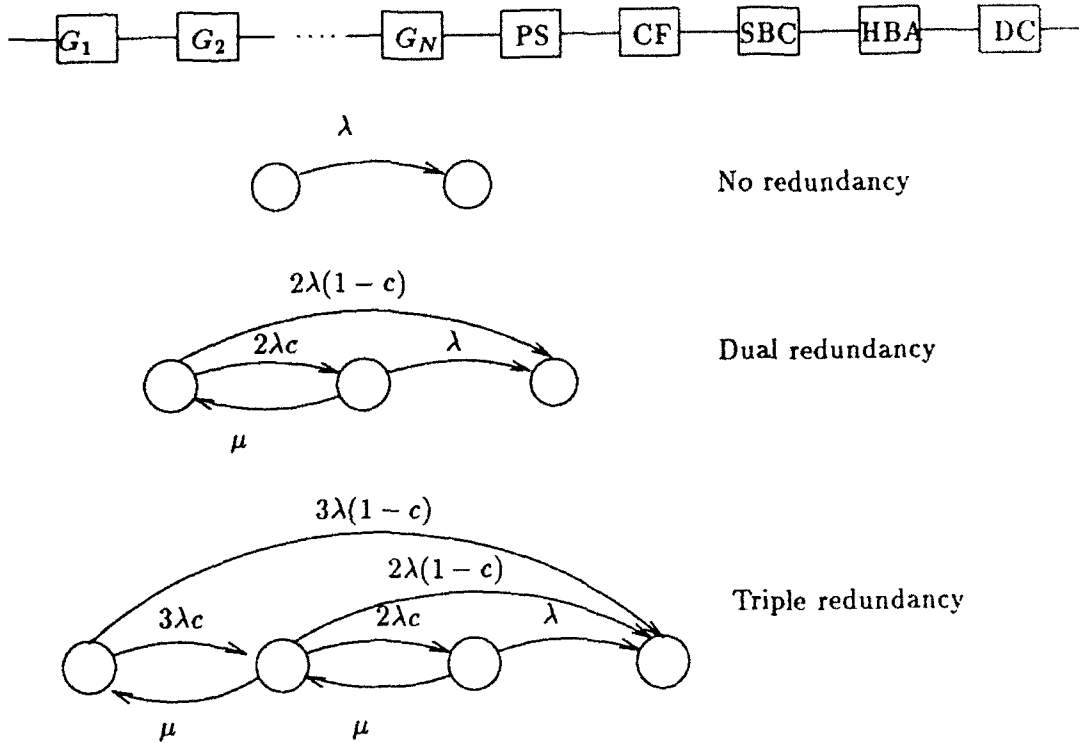


Figure 7: RAID organization with serial placement of support hardware

RBD is a simple extension of RBD shown in Figure 1. The reliability of the disk array is now given by :

$$R_{da}(t) = \left(\prod_{i=1}^N R_i(t) \right) R_{hba}(t) R_{dc}(t) R_{sbc}(t) R_{ps}(t) R_{cf}(t) . \quad (7)$$

Depending upon the number of redundant spares of each component, the reliability model for each component is different. We show Markov reliability models for components with no redundancy, dual redundancy, and triple redundancy. These models are easily extended to higher levels of redundancy. In each of these models, λ is the failure rate of the component, μ is the repair rate, and c is the coverage probability. We assume hot spares for hardware components which fail at the same rate λ .

5.2 Orthogonal Placement of Support Hardware

Schulze et al [15] proposed an organizational scheme for placement of disks and the support hardware in a way that makes disk arrays more fault-tolerant. The disks are organized in a two dimensional grid with each row representing a parity group of disks. In RAID, each parity group can tolerate single disk failures. Support hardware (power supply, cooling fans, HBA, etc.) is provided for each column of disks. Thus, each column forms a support hardware group. This organization is shown in Figure 8. This orthogonal placement of parity groups against support hardware groups provides fault-tolerance against failure of support hardware components. Disk array is operational even if all the disks in a column group or any support hardware components along a column fail. However, disk array in this case is more expensive than the serial organization because of large number of associated hardware components.

In [15], an approximate estimate for the MTTF of a disk array organized in this manner is provided. Due to complex dependence of failure and data-reconstruction in this RAID organization, a simple reliability model can not be developed. We developed a reliability model using stochastic Petri nets but it resulted in a very large Markov chain. The symmetry in this model prompted us to develop a smaller approximate model. The approximate model is obtained by essentially lumping identical states into one state. The approximate Markov model has only four states and yielded solutions that were close to the solutions obtained using the exact model.

The approximate model is shown in Figure 9. State 2 is the fully operational state of the disk array with no disk or hardware component failed. Assume that all the support hardware components are statistically independent and identical across different columns. Failure of any support hardware component in a column causes the entire column to fail. Assuming exponential time to failure distribution for each of the support hardware component, the failure rate of each hardware column λ_{sh} is the sum of failure rate of each component (CF, PS, HBA, DC, etc.) and time to failure distribution of each hardware column is exponential with this rate [16]. In state 2, one of the hardware columns may fail (transition to state 1) and it is repaired at rate μ_{sh} . For the lack of real data, we assume that each hardware

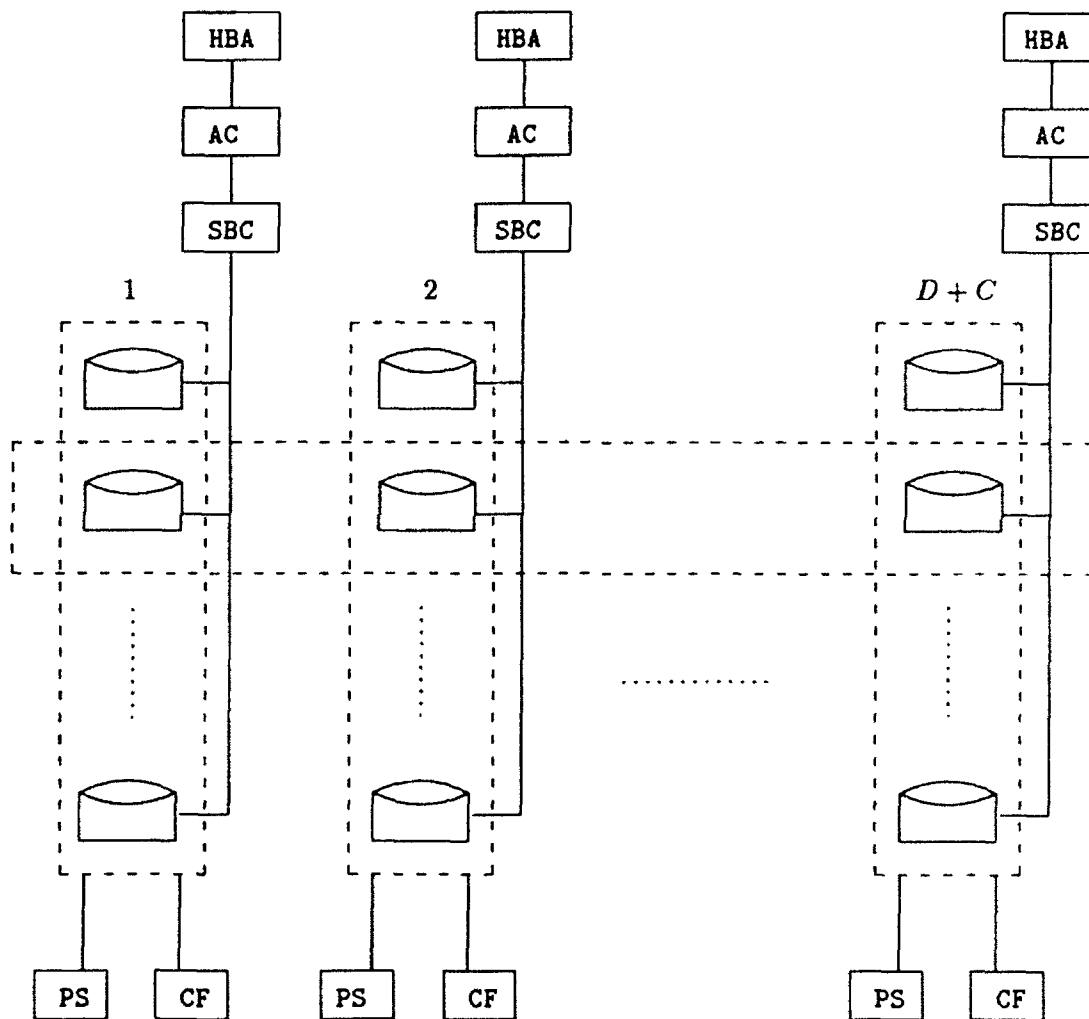


Figure 8: RAID organization with orthogonal placement of support hardware

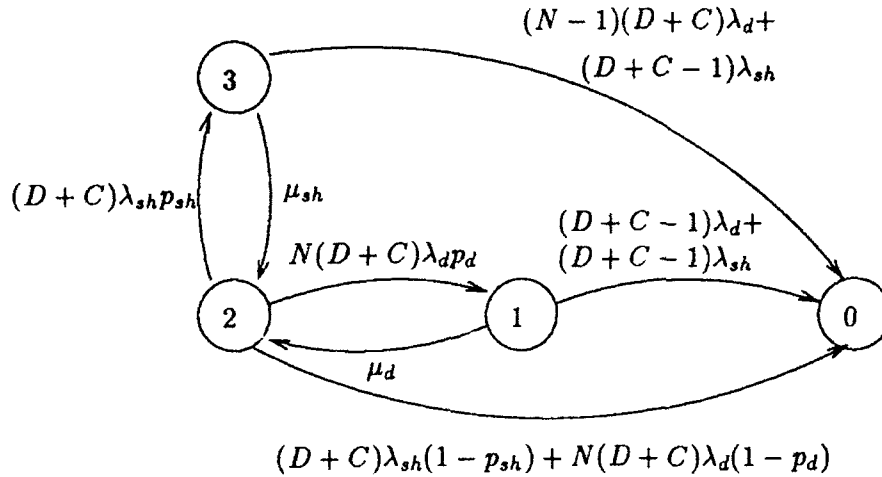


Figure 9: Approximate reliability model for orthogonal RAID-1,2,3,4,5

component has the same MTTR. If this is not the case, then simple extensions to the reliability model can be made by introducing different failure states for failures of different hardware components and their repair.

A failure in a hardware column is covered with probability p_{sh} . An uncovered failure causes the disk array to fail. While the repair is underway, the disks in this column are considered unoperational. However, if any of the remaining $(N-1)(D+C)$ disks fails or if any of other column hardware groups fail before the repair is completed, then data loss occurs (transition to failed state 0). Similarly, in state 2, any of the $N(D+C)$ disks may fail (λ_d is the disk failure rate) (transition to state 1). A disk failure is covered with probability p_d . An uncovered disk failure causes data loss. Data-reconstruction rate is μ_d . If any of the other disks in this group fail or if any of the other hardware column groups fail before the data-reconstruction is completed, then data loss occurs.

6 Numerical Results

We conducted several experiments with the models described in previous sections. It is hard to compare the different levels of RAID because it is not quite clear, for example, what

constitutes an equivalent RAID-1 organization given a RAID-5 organization. We designed our models assuming a required storage capacity of the disk array (32 disks). We also assume that each level of RAID uses identical disks (same capacity and same mean time to failure) which presumably come from the same manufacturer. Given these assumptions, we stress that the reader should lay emphasis on the characteristics of individual curves (each curve corresponds to a RAID level) and not on comparing different curves on the same plot. For instance, given a RAID-5 with 16 groups of 16 data disks each, its equivalent RAID-1 would have 256 data disks and 256 mirrored disks of same capacity. This is exorbitantly redundant. A system designer would rather implement a RAID-1 architecture with 8 data disks of larger capacity.

The base model we use for RAID-1 has 32 data disks and 32 mirrored disks. For RAID-2, the model has 8 groups each consisting of 4 data disks and 3 check disks. For RAID-3, 4, and 5, the base model has 8 groups of 4 data disks and 1 check disk. The numerical values of some of the model parameters are chosen based upon the data given in [11, 15]. The time to failure of an active disk (data and check disks) is exponentially distributed with mean 40000 hours ($\lambda = 1/40000$ per hour). We assume the distribution of time to data reconstruction is exponential with rate μ . If hot disk spares are maintained, mean data-reconstruction time is 2 hours ($\mu = 1/2$ per hour). If cold disk spares are maintained, then mean data-reconstruction time is 50 hours. If no disk spares are maintained, then mean data-reconstruction time is 74 hours. Failure rate of a hot spare disk is $\lambda_{sp} = 1/50000$ per hour. The coverage probability in each case is assumed to be 0.9

In models with predictive disk failures, rate of false alarm is chosen to be $\gamma = 1/100000$ per hour and probability that an impending failure is correctly predicted is chosen to be $\alpha = 0.9$. For models with support hardware components, we have from the data provided in [15], MTTF for power supply = 1460 hours, HBA = 123000 hours, power cable = 10000000 hours, SCSI cable = 21000000 hours, cooling equipment = 195000 hours, SBC = 40000 hours and HDD controller = 30000 hours. We take mean repair time for any support hardware component (also MTTR for any support hardware column) to be 24 hours ($\mu_{sh} = 1/24$ per hour).

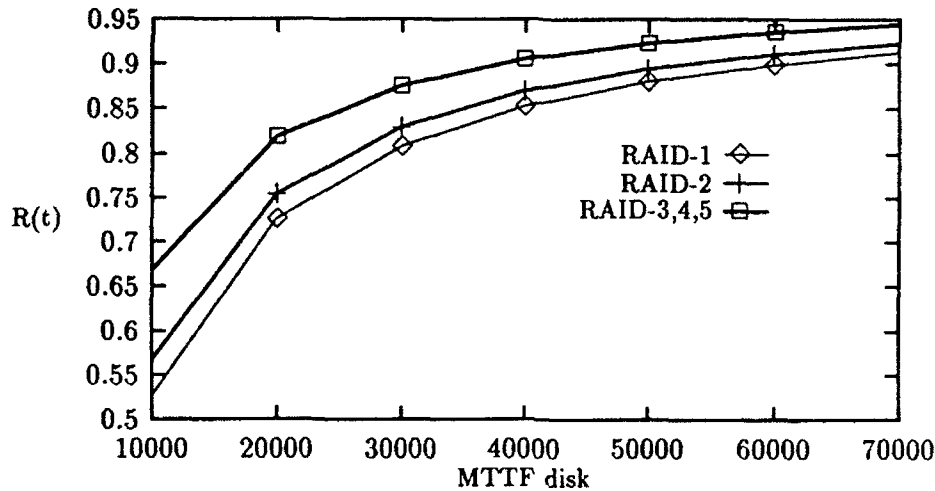


Figure 10: Reliability v/s MTTF disk in hours

All these two-level hierarchical Markov models were solved using the software package SHARPE [13].

6.1 How reliable should each disk be?

The two-level hierarchical model composed of submodels shown in Figures 1 and 2 (RAID-1,2) or 3 (RAID-3,4,5), was solved. Assume that hot disk spares are maintained in each case and a spare is available each time a disk fails. Figure 10 shows how the reliability (evaluated at $t = 1000$ hour) of various disk array architectures varies as MTTF of a single disk increases. The reliability gain is significant as MTTF of each disk is increased from 10000 hours to 40000 hours. However, the gain in reliability is not much as MTTF of a disk is increased beyond 40000 hours.

6.2 Are RAID architectures reliable enough for mission-critical systems?

Once again, the same two-level hierarchical models as the earlier experiment are solved. In Figure 11, reliability of the disk array is plotted as a function of mission time. Disk arrays are highly reliable for operation period of 500 hours or less. However, for mission-critical

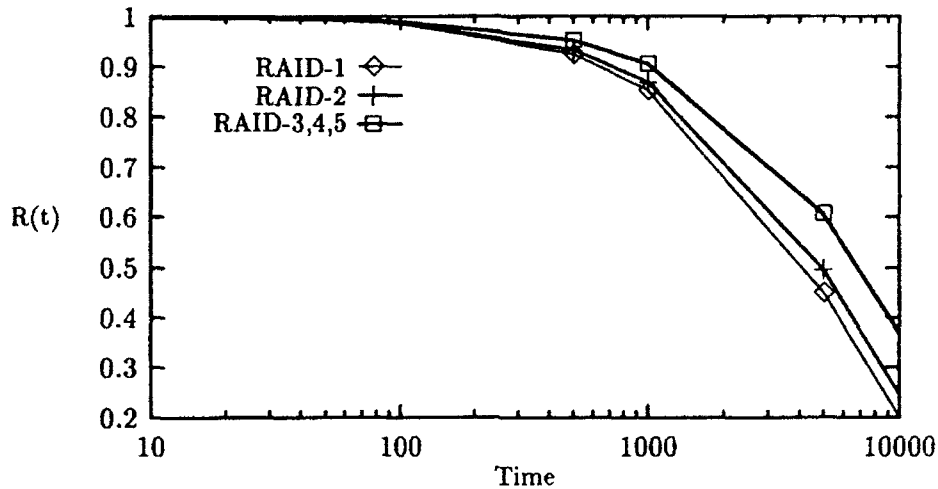


Figure 11: Reliability v/s Time in hours

systems with long mission times, we see that different disk array architectures with the given amount of redundancy are not very reliable. Note that in these models, we have not taken into account the reliability of support hardware which decreases the reliability of disk array.

6.3 How much does improved coverage of disk failures help?

We solved two-level hierarchical model composed of submodels shown in Figure 1 and 2 (RAID-2) or 3 (RAID-3,4,5). In Figure 12, reliability of the disk array (at $t = 1000$ hours) is plotted as a function of coverage probability. Tremendous improvement in reliability is achieved with improved coverage of disk failures. Whereas it is impossible to get rid of catastrophic failures, it is possible to improve the reliability of support hardware, ECC, and disk controllers by introducing redundancy. It should be noted that coverage probability may well depend upon the RAID architecture. It can be argued that RAID-1 will have higher coverage probability than RAID-3,4,5, because error detection and correction strategy of RAID-3,4,5 is more prone to uncovered failures.

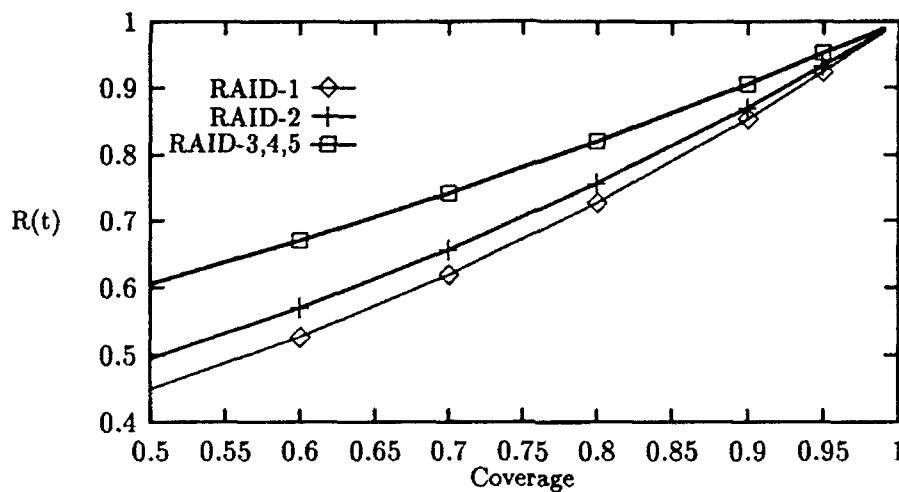


Figure 12: Reliability v/s Coverage

6.4 How low should the data-reconstruction time be?

Let us now consider the effect of mean time to data-reconstruction (MTDR) on disk array reliability. The two-level hierarchical model composed of submodels shown in Figures 1 and 2 (RAID-1,2) or 3 (RAID-3,4,5) was solved. The plots in Figure 13 show that data-reconstruction time does not significantly affect the reliability of disk system. Reliability is evaluated at time $t = 1000$ hours. Varying MTDR from 2 hours to 100 hours did not change disk array reliability much. The reason for this is because the MTTF of a disk is much larger than MTDR (mean time to data-reconstruction) of disk. Thus, taking expensive measures to reduce the data-reconstruction time would not yield significant gains. The virtual independence of array reliability on data-reconstruction time suggests the use of cold disk spares over hot disk spares. Hot spares are prone to failure just like data disks, but cold spares do not fail. Besides, it is more expensive to maintain hot spares than cold spares. However, depending upon the application, it may be useful for a variety of reasons (loss of revenue, user requirements etc.) to minimize the data-reconstruction time.

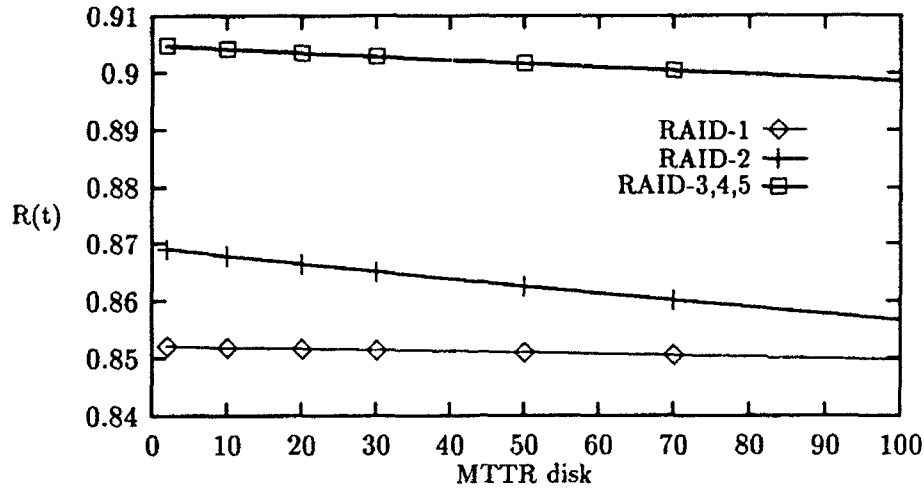


Figure 13: Reliability v/s Mean time to data-reconstruction

6.5 How many disk spares are needed?

In the previous models, we assumed unlimited number of hot disk spares were maintained. In Section 4, we saw that with some data-reconstruction cost overhead, it is possible to ensure approximately permanent availability of a disk spare. In this section, we analyze the dependence of array reliability on the number of spares and the kind of spares (hot or cold). We solved the two-level hierarchical model composed of models shown in Figure 1 and 4. (RAID-1,2 with cold disk spares) or 5 (RAID-3,4,5 with cold disk spares) or 6 (RAID-1,2,3,4,5 with hot disk spares) to analyze the dependence of MTDL on the number of spares.

In these models, we choose $\mu_3 = 1/50.0$ per hour (data-reconstruction rate when a cold spare is available), $\mu_2 = 1/74.0$ per hour (data-reconstruction rate when no spares are available), and $\mu_1 = 1/2.0$ per hour (data-reconstruction rate when a hot spare is available on site). The MTTF of a hot disk spare is taken to be 50000.0 hours ($\lambda_{sp} = 1/50000.0$ per hour), which is larger than the MTTF of an active disk (= 40000.0 hours).

For RAID-3,4,5, MTDL of disk array as a function of number of spares (cold and hot)

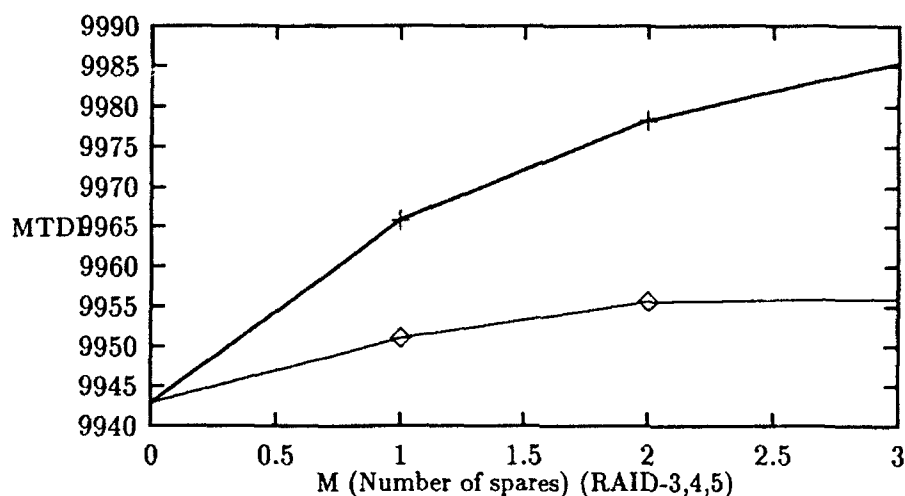


Figure 14: MTDL of RAID-3,4,5 v/s Number of spares

is increased from zero to three is plotted in Figure 14. The relative gain in MTDL is not much supporting the results of section 6.4. The absolute gain in MTDL is not much as the number of spares is increased beyond two per group. Similar trends are observed for RAID-1 and RAID-2.

6.6 Is RAID reliability scalable?

A natural step towards more parallelism in I/O transfers would be to scale the disk arrays in two dimensions. One is to increase the number of disks in a group and the other is to increase the number of groups (or both). The other reason to scale disk arrays could be simply an increased demand on storage capacity. We wish to find out if the reliability of the disk arrays scales appropriately or not. RAID-1 architecture is obviously not scalable. Increasing the number of disks reduces array reliability and MTDL. Moreover, it is not cost effective to duplicate all the disks in the system if we intend to use over 100 small disks. Thus, for RAID-1 architecture, the number of disks should be kept small and the size of each disk should be increased. Important thing to remember is that MTTF of a large disk is

not significantly lower compared to MTTF of a small disk. Thus, the above solution yields a more reliable design of RAID-1.

Given the storage capacity for RAID-2,3,4, and 5, the choice of number of data disks in a group and number of groups is dictated mainly by performance considerations (e.g., the amount of parallelism in I/O transfers) and support hardware available (e.g., the number of I/O channels, array controllers etc.). We illustrate how these choices affect array reliability. The two-level hierarchical model solved in this case is composed of submodels shown in Figures 1 and 2 (RAID-2) or 3 (RAID-3,4,5).

Given a fixed number of disks in a group ($D = 8$), the number of groups is varied. Figure 15 illustrates how reliability decreases as N increases. Next, given a fixed number of groups ($N = 8$), the number of disks in a group (D) is varied. Figure 16 shows how reliability decreases with increase in D . Reliability in both the cases is evaluated at time $t = 1000$ hours. These plots reveal that reliability of all the RAID architectures falls below acceptable levels as the disk arrays are scaled up in dimensions. A simple solution is to scale the redundancy in hardware as the dimensions of a disk array are scaled. Another solution would be to come up with newer designs of RAID with more fault-tolerance like the one suggested in [2].

6.7 How much do we gain by orthogonal placement of support hardware?

We now analyze the gains in reliability due to orthogonal placement of support hardware components over the serial placement. Assume that there is no hardware redundancy in either organization. For serial organization, the two-level hierarchical model composed of submodels shown in Figures 7 and 2 (RAID-1,2) or 3 (RAID-3,4,5) is solved. For orthogonal organization, the model shown in Figure 9 is solved. The array reliability is plotted as a function of time in Figure 17.

Comparing the two curves (eg., RAID-1 (srl) and RAID-1 (otg)), we find that orthogonal placement of support hardware improves array reliability. A key pattern to note is that for orthogonal organization, RAID-1 architecture has higher reliability than RAID-3,4,5 or RAID-2 (as opposed to all the earlier plots). This happens because there are only two

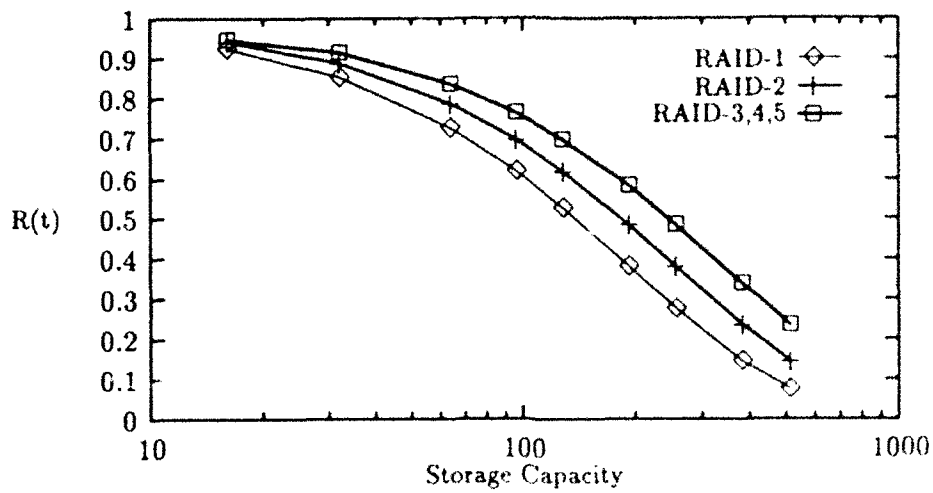


Figure 15: Reliability v/s Storage Capacity ($D = 8$ for RAID-2,3,4,5)

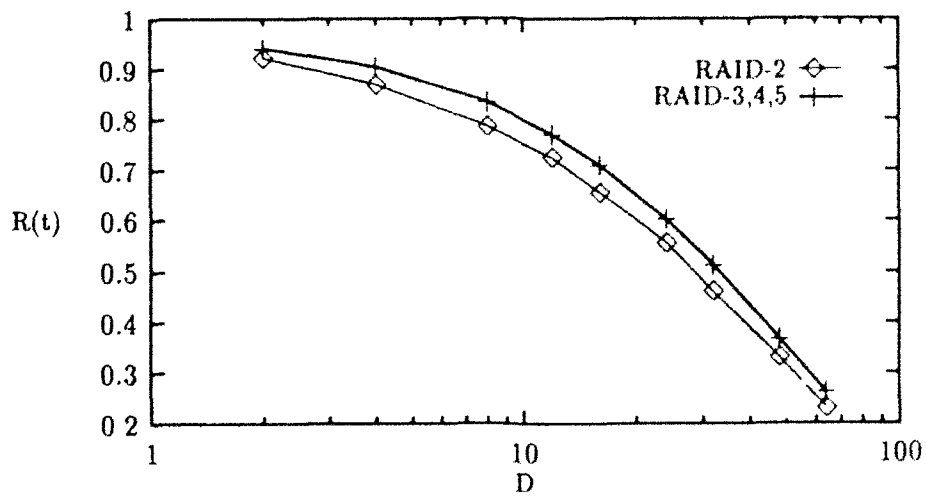


Figure 16: Reliability v/s D ($N = 8$) (RAID-2,3,4,5)

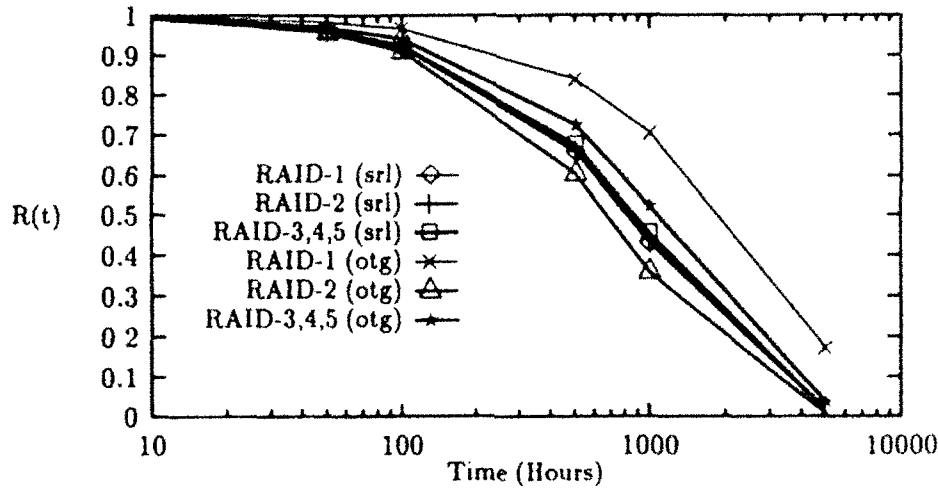


Figure 17: Serial RAID reliability as a function of time (hours)

hardware columns in RAID-1. Thus, RAID-1 benefits the most from orthogonal placement as far as improvement in reliability is concerned. Moreover, the overhead cost due to multiple support hardware components (one for each column) of RAID-1 is the least among different RAID organizations. In serial placement of hardware, the reliabilities of different RAID architectures are almost the same. This happens because of the very small MTTF of support hardware (≈ 1460 hours) compared with the MTTF of disks.

7 Conclusion

We have carried out reliability analysis of different fault-tolerant disk array architectures classified as different levels of RAID. The reliability models formally capture the operational dependency of disk array system on array organization and support hardware. Solution of these models provides useful insight into the dependence of disk array reliability on parameters such as MTTF of disks, mean data-reconstruction time, coverage of faults, and dimensions of disk arrays.

If the disk array is intended for a mission-critical application with long mission times,

then it must possess high reliability over a long period of time. Our results show that none of the RAID architectures meet the ultra-high reliability requirements in their present implementation. However, introduction of additional redundancy for key components may help achieve desired reliability. If the MTTF of a single disk is increased beyond a certain value, the gains in disk array reliability are not significant. Thus, ultra-reliable individual disks are not the solution for ultra-reliable disk arrays. Reducing data-reconstruction time does not yield significant array reliability gain either. However, tremendous improvement in disk array reliability can be obtained with improved coverage of faults (i.e., improved fault-detection and more reliable support hardware). Thus, the key to improving disk array reliability is superior fault coverage. Dimensional scaling of disk arrays results in reliability degradation. Therefore, hardware redundancy must be scaled as the dimensions of disk arrays are scaled to maintain high reliability.

If reliability of support hardware (Power supply, cooling hardware, array controller, host bus adaptor etc.) is taken into account, then overall reliability of disk array decreases. Orthogonal placement of support hardware increases the overhead cost but significantly improves the reliability. The power supply is the bottleneck of support hardware reliability and therefore of disk array reliability. The best gains in reliability are achieved if repair times for support hardware are reduced. One way of achieving this is to maintain spares for each component.

We expect these models and results to be useful to designers of disk array architectures during the design as well as operational stages since these models reveal the bottlenecks in array reliability. Appropriate steps could be taken either by modifying the array design during the design stages or by introducing additional hardware redundancy if the system is already in operation. Given specified reliability requirements, these models also test whether a given disk array architecture meets those specifications or not.

References

- [1] D. Bitton and J. Gray. Disk shadowing. In *Proc. Very Large Database Conf.*, pages 331-338, Long Beach, Calif., Aug 1988.

- [2] G. Gibson, L. Hellerstein, R.M. Karp, R.H. Katz, and D.A. Patterson. Failure correction techniques for large disk arrays. In *Proc. Third Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, April 1989.
- [3] G. A. Gibson. Performance and reliability in redundant arrays of inexpensive disks. In *Proc. of CMG conference*, pages 381-391, Chicago, IL, Dec 1989.
- [4] G.A. Gibson and D.A. Patterson. Designing disk arrays for high data reliability. Technical Report CMU-CS-92-130, Carnegie Mellon University, April 1992.
- [5] R.W. Hamming. Error-detecting and correcting codes. *Bell Systems Tech. Journal*, 26:147-160, April 1950.
- [6] R.H. Katz, G.A. Gibson, and D.A. Patterson. Disk system architectures for high performance computing. *Proceedings of the IEEE*, 77(12):1842-1858, 1989.
- [7] M.Y. Kim. Synchronized disk interleaving. *IEEE Trans. on Comp.*, C-35:978-988, Nov 1986.
- [8] M. Livny, S. Khoshafian, and H. Boral. Multi-disk management algorithms. In *Proc. SIGMETRICS*, May 1987.
- [9] W.E. Meador. Disk array systems. In *Spring COMPCON 89*, pages 143-146. San Francisco, CA, March 1989.
- [10] A. Park and K. Balasubramaniam. Providing fault-tolerance in parallel secondary storage systems. Technical Report CS-TR-057-86, Dept. of Computer Science, Princeton University, 1986.
- [11] D. Patterson, P. Chen, G. Gibson, and R. Katz. Introduction to redundant array of inexpensive disks (RAID). In *Proc. COMPCON 89*, pages 112-117, 1989.
- [12] D. Patterson, G. Gibson, and R. Katz. A case for redundant array of inexpensive disks (RAID). In *ACM SIGMOD conference proceedings*, Chicago, IL, June 1988.
- [13] R. A. Sahner and K. S. Trivedi. Reliability modeling using SHARPE. *IEEE Trans. Reliability*, R-36(2):186-193, June 1987.
- [14] K. Salem and H. Garcia-Molina. Disk striping. In *Proc. IEEE Data Engr. Conf.*, Los Angeles, CA, Feb 1986.
- [15] M. Schulze, G. Gibson, R. Katz, and D. Patterson. How reliable is a RAID ? In *Spring COMPCON 89*, pages 118-123, San Francisco, CA, March 1989.
- [16] K.S. Trivedi. *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. Prentice-Hall, Englewood-Cliffs, NJ, 1982.

DESIGN SYNTHESIS METHODS

RAPID PROTOTYPING

Janet Y. Higgins
Naval Surface Warfare Center, Dahlgren Division
Silver Spring, MD 20903-5000

INTRODUCTION

Navy systems are becoming increasingly complex. Development of these systems will require changing the methods by which these systems are designed. This change should include the integration of prototyping into the entire design cycle. Currently, prototyping is often delayed until many critical design decisions have been constrained. This leads to inferior systems because many of these design decisions were not optimal. Through the integration of prototyping into the design cycle, the system designer has increased capabilities to check the feasibility of specifications and requirements, compare the performance of alternate design choices through trade-off analysis of hardware, software, or humanware implementations, or to consider the performance of different algorithms. Rapid prototyping is necessary for prototyping to be a practical aid in the design cycle.

PROTOTYPING USEFULNESS

There are two major reasons for prototype development: to demonstrate proof-of-concept for high risk sections of a system and to assist in the development of the final product. Successful integration of prototyping into the system design process requires that system designers and management understand the differences between these two purposes.

The proof-of-concept prototype is usually classified as a "throw-away" prototype. It is developed to answer questions about one particular high-risk section of a project. In general, as shown in figure 1, the greater the risk associated with the prototype, the more likely the prototype will be thrown away. This prototype aids the developers either in determining the feasibility of an approach or by allowing exploration of the best method by which to solve a problem. Proof-of-concept prototyping must be done early in the design cycle because the answers it provides will usually drastically shape the final system. Also, the proof-of-concept prototype can show the proposed system is not feasible and thus signify that a major review of the system concept is required. Two dangers affect the development of the proof-of-concept prototype. First, management may oppose the expenditure of resources to develop a prototype which will be thrown away. This view deprives the developers of the knowledge and experience gained through proof-of-concept development. Secondly, management and designers may be tempted to incorporate the prototype into the final product. However, proof-of-concept

Characterizing Prototyping Processes A Dimension: Degree of Experimental Intent

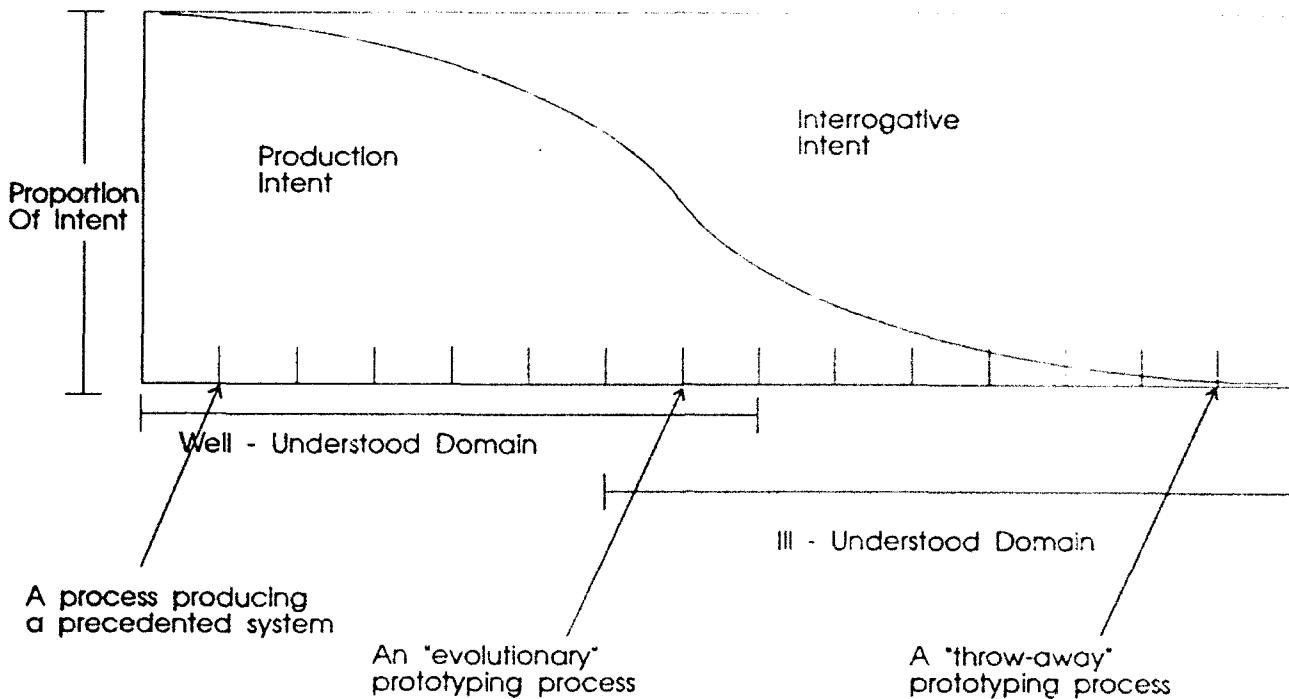


Figure 1. (Proceedings: Spring 1992 Prototech Community Meeting)

prototypes are generally not designed for maintainability, fault-tolerance, reliability, or security.

Those prototypes which aid in the development of the final product are usually classified as "evolutionary" because they often gradually evolve into the product. Early in the design cycle, they can clarify requirements, determine the sufficiency of the requirements, and facilitate communication between system designers and management. Later in the design cycle, they can allow trade-off analysis between different design decisions. Finally, once portions of the system are implemented, these portions can often be integrated into the prototype to allow better performance measurement and prediction. Communication, trade-off analysis, and performance measurements are some of the major strengths of evolutionary prototyping. Caution must be used; however, to insure that the system is designed for maintainability, fault-tolerance, and reliability.

PROTOTYPING APPROACHES

To meet the purposes of proof-of-concept and product development prototypes, several different approaches have emerged. Examples of products are described to illustrate the different prototyping approaches. For certain approaches, many other similar products also exist other than those described.

Simulation - Simulation uses software to model components of the system. These components can be hardware, software, or humanware. Simulation provides an ability to measure performance, perform rapid trade-off analysis, and integrate actual pieces of software as they are developed. Disadvantages of simulation are that large development efforts can be required, fairly detailed designs can be required, and actual hardware usually can not be used.

IDAS - Integrated Design Automation System, JRS Research Laboratories, Inc. IDAS provides the tools necessary to map a particular program onto a given architecture, to measure the quality of that mapping, and to detect where and under what conditions that mapping fails to be optimum. IDAS provides quick and efficient evaluation of design alternatives, execution of desired benchmarks on simulators customized for specific machine designs, and allows the designer to iteratively improve that design until the desired level of near optimum solution has been reached. From this architecture, trade-off analysis can be performed to determine an architecture which satisfies non-functional restraints such as size, cost, weight, and power.¹

SES Workbench - This product allows a system to be modeled as a collection of components such as processors, resources, and delays. From this system, information such as timing, resource contention, and program design can be obtained. It provides the ability to perform rapid trade-offs in components and in their configuration.²

Prototyping languages - These languages provide the designer with the ability to work at a higher level of abstraction than previously available to describe their domain. Much research is being done to develop the higher level prototyping languages. The hardware description languages are well defined and in use; whereas, Proteus and other high-level prototyping languages are still in the experimental stage.

VHDL - VHSIC hardware description language supports the design, description, and efficient simulation of VHSIC components. Its ability to describe hardware at various abstractions from the high-level, systems-oriented view down to the gate level make it appropriate for prototyping. VHDL is the DOD standard for describing VHSIC hardware.³

Verilog - This is an alternate VHSIC hardware description language. Similar to VHDL, it is becoming popular in the development of commercial products.

Proteus - Proteus provides an extensible set of high-level architecture-independent primitives for parallel and distributed computation; a data model supporting algebraic specification; and an identification of specifications and types which supports object-oriented notions of subtyping and inheritance. It

provides a formal concept of module refinement whose implementation supports both the evolutionary model of software development, through the refinement of control and data-abstractions to improve the efficiency of early prototypes, as well as architectural targeting by refining architecture-independent prototypes into restricted forms targeting execution on specific parallel platforms.⁴

Software/hardware module interconnection schemes - This approach is based on maintaining a library of components and having the ability to connect them together to form the system. This is one of the principal areas of prototyping research.

CAPS - Computer-Aided Prototyping System, Naval Postgraduate school. The CAPS method uses a prototyping language to design a hierarchical system structure with real-time and control annotations and automatic code generation together with reusable components to produce executable prototypes.⁵

PERTS - A Prototyping Environment for Real-Time Systems, University of Illinois. PERTS will provide an environment for the use and evaluation of new design approaches, for experimentation with alternative system building blocks, and for the analysis and performance profiling of prototype real-time systems.⁶

NSS - Network Synthesis System, JRS Research, Inc. This tool provides the ability to rapidly prototype a network and evaluate it against its specifications and constraints. The tool will rely on a reusable parts library for both software (Ada) and Hardware (VHDL).⁷

User interface developers - Environments which support user-friendly interface development are extremely important for communication between the designer and management. A generally accepted practice is for designers and management to walk-through the user interface screens before much of the functionality has been implemented. At these walk-throughs, errors in requirements or requirement understanding are often detected. Many commercial products exist which support user interface development.

RESEARCH AREAS

Although many promising approaches to prototyping are currently available, there remain many critical technologies which must be developed in order to comprehensively support the design cycle. Some of these areas are listed below.

Module Interconnection Formalisms - These formalisms will allow the development and management of software databases and the

mechanism for interconnecting existing software independent of the language used to implement it, the platform it runs on, or the communication media available to access it. With the many research efforts emphasizing prototyping through module interconnection, it is important for these formalisms to be developed and adopted in the near future. Currently, the DARPA Prototech community is investing this issue.⁸

Information Abstraction - To allow analysis at a high-level and simplify lower level analysis, information is abstracted away. This presents a danger that the information will lose its identification with reality. Without the ability to reference reality, the validity of the prototype is difficult to determine because it is no longer possible to determine the validity of the initial information. Also, information abstraction currently hides timing and resource utilization information limiting the usefulness of prototyping to complex, real-time applications.⁹

Prototype migration - Current techniques do not permit the migration of one design phase prototype to the next phase. This limits the integration of prototyping in the design process because of the effort required to prototype at each stage and the inability to support a spiral design cycle since information can not easily be passed between phases.

CONCLUSIONS

Future Navy systems will require a departure from current system development practices if they are to have higher performance; better reliability, fault-tolerance, security, and maintainability; faster development time; and lower cost. One promising approach to improve design methodology is to integrate rapid prototyping into the design cycle. For this approach to succeed, designers and management must understand the different uses for prototypes. Currently many developers do not distinguish between different types of prototypes and as a result, prototypes are viewed as a great expense with little return. Additionally, the integration of prototyping into the design cycle will require the development of tools and techniques that permit the designer greater flexibility and faster results than he has using current methodologies. Although prototyping has great promise, successful research in interconnection formalism, information abstraction, and prototype migration will greatly enhance the current capabilities of prototyping.

REFERENCES

1. "Software User's Manual for the Integrated Design Automation System", JRS Research Laboratories, Inc. December 11, 1989.
2. Jenevein, Roy, et. al., "Critical Component Analysis and Synthesis for Hardware/Software Systems", Proceeding of the Systems Evaluation and Assessment Technology Workshop, Naval Surface Warfare Center, 20 - 22 August, pp. 35 - 39.
3. "VHDL User's Manual: Volume 1 - Tutorial and User's Guide", U.S. Air Force Technical Report IR-MD-029, July 30, 1984.
4. Duke/UNC/Kestrel, Team Progress Report, Proceedings: Spring 1992 Prototech Community Meeting, Honeywell SRC Technical Report CS-M92-001, May 27 - 29, 1992.
5. Luigi, V. Berzins, U. Kodres, and Y. Lee, "Prototyping a Low Cost Tactical Display System", Proceedings of the Systems Evaluation and Assessment Technology Workshop, Naval Surface Warfare Center, 20 August 1991, pp. 23-30.
6. Liu, J. W. S, K. J. Lin, and C. L. L' "PERTS: A Prototyping Environment for Real-Time Systems," Proceedings of the 1991 Systems Design Synthesis Technology Workshop, Naval Surface Warfare Center, 10 September 1991, pp. 137-149.
7. Warshawsky, E. H., "The Network Synthesis System", submitted to 1992 Complex Systems Engineering Synthesis and Assessment Technology Workshop. Naval Surface Warfare Center. July 20 - 24, 1992.
8. Purtilo, et al., "Module Interconnection Formalism: A Coloring Book," Draft version, Spring 1992 Prototech Community Meeting, May 27 - 29, 1992.
9. Baker, Ted, "Prototyping Real-Time Systems: Some Practical Considerations," Proceedings: Spring 1992 Prototech Community Meeting, Honeywell SRC Technical Report CS-M92-001, May 27 - 29, 1992.

A NEW PARADIGM FOR THE DESIGN AND IMPLEMENTATION OF LARGE-SCALE DISTRIBUTED REAL TIME SYSTEMS

Insup Lee and Noah Prywes
Computer Command and Control Company, Philadelphia, PA 19103
and
Department of Computer and Information Science
University of Pennsylvania, Philadelphia, PA 19104-6389

ABSTRACT

The paper considers the problem areas that need to be investigated to develop a new real-time system design paradigm for the emerging applications and computing environment. The envisioned environment will use the so called *intelligent networks* comprised of processors, storage, communications and input/output resources and a methodology for their management. It will be shared by multiple applications which will be guaranteed to be executed with specified timing and reliability. These applications and the computing environment will be so extensive that use of automatic tools that involve simulation and optimization of allocation of resources will be imperative. While much research has been conducted on system specification languages, far less effort has been expended on how the systems specified using these languages can be evaluated and analyzed. While basic techniques have been developed in the respective areas, it is necessary to develop a consistent and practical set of models of the entities in the application and environment and the network management methodology. These models will then have to be used practically in simulating and optimizing the respective designs. The collection of these models will in fact constitute the simulation and optimization tools. The paper focuses on the following: modelling of concurrency methods; Simulation models for application entities, simulation models for intelligent network entities, simulation of application timing, simulation models of recovery, and optimization models

Basic techniques have been proposed in the above areas. It is necessary to integrate them in an overall practical simulation and optimization guided by human design. These models will provide also a consistent framework for formal definitions of the semantics of the application and computing environment entities used in specification languages, including the interactions between the entities.

1. INTRODUCTION

The emerging computing environment is envisaged as consisting of high-performance computing resources connected through a high-speed communications network which execute diverse applications in a timely and reliable manner. The effective exploitation of the full potential of such an environment will require an advanced software/hardware design paradigm based on sound methodologies, use of computer-aided tools, and education of the designers in the new paradigm. [52]

The key aspect of the paradigm is providing throughout the system's life cycle simulation and optimization support for intermediate designs, and interactive man-machine redesign by feedback of the results. The paper discusses the problem areas in developing this paradigm.

The paradigm focus is on real-time geographically distributed applications. Typical applications are manufacturing systems, monitoring systems, weapons systems and intelligent communication systems. Currently, most of these applications use dedicated networks. However, in the envisaged environment, these networks will be shared by many applications. Examples are provided in the *intelligent networks* envisioned by commercial telephone companies. [1,45] They will support many virtual networks with guaranteed performance and reliability, each virtual network supporting its own time-critical application.

The potential high cost associated with an incorrect operation of these systems demands a rigorous framework in which design alternatives can be postulated, optimized, and analyzed during design as well as during their operations (to react to dynamic changes in the computing environment). These systems are costly to prototype as they change dramatically. Thus the timing properties of design alternatives must be carefully evaluated under varying conditions. The design of these systems is becoming increasingly difficult because more functionalities are expected and more distributed components are networked together. Because of this, the role of the human designer is limited to providing guidance in exploring alternatives that can be expressed on the high level of system architecture. Because of the complexity of the applications, design alternatives can only be effectively evaluated and optimized automatically through extensive simulation.

Three major criteria must be applied to the methods that embody the paradigm:

- a) The man-machine interactions required to guide the design and propose alternatives.
- b) The evaluation of system entities and their interactions through simulation.
- c) The optimization of system architecture and resource allocation on a dynamic basis.

There has been much research on specifying the applications. As discussed below, we accept a specification language as a given. Our objective is to outline the evaluation and analysis models for applications specified in these languages which are executed using an intelligent network. The consistent collection of these models will constitute the practical simulation and optimization tools used in the design paradigm.

The rest of the paper is organized as follows. Section 2 provides a further overview of the future operating environment and use of the design paradigm, Section 3 identifies necessary research and development areas needed in the models incorporated in design tools. It explains what needs to be done in each of these areas. Section 4 concludes the paper.

2. FUTURE IMPLEMENTATION ENVIRONMENT AND USE OF DESIGN PARADIGM

Based on current technological advances and customer expectations, it is not difficult to imagine a future high-speed intelligent network that simultaneously supports many real-time applications. An example is the intelligent telephone network that will provide world-wide distributed computing services in real-time. [1,45] There are many factors that affect the performance of such systems: the hardware resources of the distributed network, the software structure of the distributed application, the mapping of the software components to the hardware resources, and the reconfiguration capability for adapting to dynamically changing conditions.

Because of complexity, a time critical system must be developed following a design methodology that supports an iterative and top-down development. This design methodology must be based on a framework that supports the progressive re-architecture of system components and the comparative evaluation of intermediate designs. For example, the designer may describe the system as consisting of a relatively few communicating components whose internal details are hidden. The designer then uses the tools to evaluate and optimize this design assuming the existence of some hardware resources. Based on the evaluation, the designer may change its design or refine it by detailing the internals of some or all components. This will be continued until the design is detailed enough to be subjected to correctness verification. During refinement, some of earlier design decisions may also have to be incrementally revised. Above all, much of this design methodology must be supported by computer-aided tools that can automatically find optimal mappings between software components to hardware resources and simulate their performance. The simulation should also identify bottlenecks and suggest how to improve performance.

3. PROBLEM AREAS IN DEVELOPING THE DESIGN METHODOLOGY

Much research has been reported on specification of computerized systems, but relatively little has been done on the three problem areas stated above: (a) the user guiding the automated design, (b) the consistent modelling and simulation of application/resource entities, and (c) optimization of the allocation of computing resources.

There is however a considerable reservoir of basic applicable techniques in these areas that need to be investigated and adopted, as appropriate. [6,9,24,25,31,32,35,39,42,46,48]

3.1 Specification Methods

The methods of specification of computer systems reported to date may be accepted as given. [2,5,8,17,20,29,31,32,49,50] They can be generally characterized as follows:

Independence of specification of the application from specification of the computing resources: These two types of specifications are composable separately and independently. There is, however, a third specification that ties application entities with computing resources—that of timing aspects of the application entities when using available computing resource architectures.

Graphic representation of a specification: A specification can be represented as an Entity-Relation graph, where entities are nodes and relations are edges. A variety of attributes may be associated with the nodes and edges as follows:

Application specification: typically consists of objects and transformation entities (nodes) and data flow or object oriented relations (edges). [13,19] Also, it may include as attributes throughput execution times, deadlines and response times, geographical constraints, as well as reliability and recovery.

Computing Resources specification: typically consists of processing, storage, and communication entities (nodes), and interconnection relations (edges). Also, it includes attributes of processing capacities and dynamic network management and scheduling strategies.[3,4,7,46]

Hierarchically structured entities: The applications and computing resources considered here will typically be very extensive. For a human to provide design guidance, it will be necessary to create entities on multiple levels of detail, each representing an abstraction of the lower level entities. The design may be performed on a selected level of detail. Design based on a higher level is easier to specify, evaluate and verify. However, it hides lower level details. Working with a design based on higher level entities incurs the penalty of reduced precision, as well as reduced possibilities for optimization. Based on feedback from the design process, the user may have to modify the hierarchical structure by merging or subdividing entities. [40,52]

While specification languages vary, it is assumed that they may be transformed into a form that uses the models of various entities and relations discussed below. The amalgamation of these models will be the simulation and optimization tools that will be used in the design paradigm.

3.2 Design Areas That Need to be Developed

Six design areas are outline below. The state of the art of relevant techniques must be investigated and, if appropriate, incorporated into the simulation and optimization tools used in the design paradigm. They need to be evaluated based on their impact on: (a) user interactions, (b) modelling of entities for the simulation and, (c) the optimization of allocation of resources. Further, there is a need to verify the method in test applications to ascertain its practicality. The design areas are as follows:

MODELING OF CONCURRENCY METHODS: Concurrency is the prime method used to parallelize operations in order to satisfy real time requirements. A number of techniques are

available for synchronizing and scheduling concurrent distributed processes. [27,37,41] To guide the design, the user must understand the respective methods. Methods based on high level entities (such as those based on messages) may be preferred, because of ease of human interaction and verification, upon methods based on low level entities (such as those based on semaphores). The selected concurrency methods must be used to drive the simulator. They also impact the search for optimal allocation of resources and the need by the user to modify the architecture of the application based on results of the simulations. Thus, the selection of concurrency method is critical as it cuts across the entire design paradigm.

SIMULATION MODELS OF APPLICATION ENTITIES: These models actually represent the semantics of the entities expressed in the specification language to specify the application. The user must understand these semantics in order to control the progress of the design. The models must be defined for all the software objects that correspond to application entities and for the interactions between these objects. Some entities may be specified on a higher level than that of software objects and the selection of how they are to be implemented may be left to a later stage. As noted entities may be hierarchically structured. The exploding or imploding of entities must follow modelling rules as well. [15,36,43]

SIMULATION MODELS OF COMPUTING RESOURCES AND COMMUNICATION ENTITIES: These models must be defined for all the types of entities that are used: processors, storage, communications and input/output devices. Separate models are needed for entities with different architectures. Resource entities may also be hierarchically structured. A methodology is needed for exploding or imploding resource entities in a multiple level structure. In addition, alternative management methods of the computing environment must be modelled, including the scheduling and recovery algorithms. These environment management methods will be used to drive the simulation as well.

SIMULATION MODELS FOR TIMING OF APPLICATION ENTITIES: These are models for evaluating the delay involved in operations of an application entity, when using a feasible resource architecture. A delay may vary depending on the type of operation and on the state of the entity. There are a number of methods for specifying (and simulating) timing of entities. They vary in the degree of complexity in specifying the delays and the simulation time required to determine the delays.

SIMULATION MODELS FOR RECOVERY: These are models of the computing environment management for detecting and responding to events if resources malfunction and if load variations imperil the ability of the network to satisfy real time requirements. These models must incorporate evaluation of recovery time. The simulation model for occurrence of the need for recovery must be included as well. The model must take into account the effect of built-in redundancy in network resources on need for recovery.

OPTIMIZATION METHODS: The optimization is typically used to allocate resources which minimize operating costs, delays, or recovery time, or some combination of these. The envisaged applications and the computing and communication resources are extensive. The allocation space—the feasible combination of allocated resources and application entities—is immense. Manual resource allocation is totally impossible. However, the rationale in the search of the allocation space must be explainable to the user, so that user may offer guidance in the search. Resource allocations have varying life times and stabilities. For instance, distribution of information in data bases may not need to be changed for a long time. Such allocations may be taken as static and completely stable. However, the design must explore wide classes of possible allocations of communication and processors as such allocations may have to be changed frequently. Some allocations may have to be reoptimized frequently during daily operation, as the application traffic changes and as resources increase or decrease. The optimization method must be progressive so that if time allows it can continue in trying to improve the allocation of resources, and otherwise it accepts intermediate results. [11,16,26,27,30,34,40]

While basic techniques in the above areas exist, there is no systematic structure for incorporating and modifying various types of models. A comprehensive framework must be constructed in which these models can be incorporated. The productivity of the overall simulation and optimization must be tested using realistic examples [38] in the context of existing workstations and high speed processors.

4. CONCLUSION

The problem areas enumerated in this paper, while they have been stated in terms of their use by the tools of a system design paradigm, really concern the definition of basic concepts used in system specification and implementation. They provide a basis for a formal definition of the semantics of application and resource entities. They will also take into account realistically the human capacity to guide the design process and the compiling resources needed for design.

5. REFERENCES

1. M. Bahl, J. Daine, R. O'Grady, "The Evolving Intelligent Interchange Network—An SS7 Perspective," Proc. of the IEEE, pp. 637-643, April 1992
2. A. Benveniste, G. Berry "The Synchronous Approach To Reactive and Real-Time Systems", Proc. of IEEE, Vol. 79, No. 9, September 1991.
3. R. Bianchini, Jr., J.P. Shen, "Interprocessor Traffic Scheduling Algorithm for Multiple-Processor Networks," IEEE Transactions on Computers, Vol. C-36, No. 4, April 1987
4. S. R., Biyabani, J. A. Stankovic, K. Ramarathan, "The Integration of Deadline and Criticalness in Hard Real-Time Scheduling," Proceeding of the Real-Time Systems Symposium, December 1988
5. F. Boussinot R. deSimone "The ESTEREL Language," Proc. of IEEE, Vol. 79, No. 9, September, 1991, pp. 1293-1304
6. A.K.Cheng,, S., Mok C. Browne, "Computer Aided Design of Real-Time Rule-Based Systems," IEEE Trans. on Software Engineering, 1992
7. S., Cheng, J. A. Stankovic, and K. Ramarathan, "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems", Proceedings of the Real-Time Systems Symposium, New Orleans, LA, December 1986
8. E.M. Clarke, D.E. Long, K.L. McMillan, "A Language for COMpositional Specification and Verification of Finite State Controllers," Proc. of IEEE, Vol. 79, No. 9, September, 1991 pp. 1283-1292.
9. Computer Command and Control Company, Software Engineering Environment for Parallel/Distributed Systems, Final Report, Contract #N6092189-C0127, October 30, 1990
10. Z. Cvetanovic, "The Effects of Problem Partitioning, Allocation and Granularity on the Performance of Multiple-Processor Systems", IEEE Transactions on Computers, Vol. C-36, No. 4, April 1987
11. L. Davis, Ed., Handbook of Genetic Algorithms, Van Nostrand Reinhold, 1991.
12. R. Davis,, "Meta-rules: Reasoning About Control", Artificial Intelligence 15(3), 1980.
13. T. Demarco, Structured Analysis and System Specification, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987.
14. L. Erman, F. Hayes-Roth, "The Hearsay-II Speech-understanding System: Integrating Knowledge to Resolve Uncertainty," ACM Computing Surveys 12(2), 1980.
15. G.A. Frank, "Symbolic Simulation of Structured Analysis Models. An Approach to Executable Specifications," Proc. of the Systems Evaluation and Assessment Technology Workshop, NSWC, Silver Spring, MD, August 1991.
16. D.E. Goldberg, Genetic Algorithms in Search Optimization and Machine Learning, Addison Wesley, 1989.
17. N. Halbwachs, P. Cospi, R. Raymond, D. Piland, "The Synchronous Dataflow Programming Language LUSTRE," Proc. of IEEE, Vol. 79, No. 9, September 1991, pp. 1305-1320.
18. Harel, D., "Biting the Silver Bullet", IEEE Transactions on Computers, January 1992.

19. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, Shmueli-Trautring, M. Trakhtenbrot. STATEMATE: A working environment for the development of Complex reactive systems. IEEE Transactions on software engineering, 16(4) 403-414, April 1990
20. D. Hatley, and I. Pirbhai, Strategies For Real-Time System Specification, Dorset House Publishing, New York, 1987.
21. S. Hekmatpour, D. Ince, Software Prototyping, Formal Methods and YDM, Addison-Wesley, 1988.
22. C. A. R. Hoare, "Communicating Sequential Processes," CACM, Vol. 21, No. 8, pp. 666-677, August 1978.
23. S. Howell, Evaluation Results Report For Next Generation Computer Resources Operating Systems Interface Baseline Selection, NSWC, Silver Spring, MD, May 1990
24. S. Howell, P. Hwang, C. Nguyen, M. Trinh, Destination Allocation Algorithm Library and Source Code, Naval Surface Warfare Center, Technical Report, February 1992
25. S. Howell, P. Hwang, C. Nguyen, "Expert Design Advisor", Proc. 5th Jerusalem Conference On Information Technology, IEEE Computer Society Press, Los Alamitos, CA, October 1990, pp 743-756.
26. S. Howell, C. Nguyen and P. Hwang, "Design Structuring and Allocation Optimization", Proc. Hawaii International Real-Time Systems Conference, January 1992
27. T. Ibaraki, and N. Katoh, Resource Allocation Problems: Algorithmic Approaches, MIT Press, Cambridge, MA, 1988.
28. L.H. Jamieson, D. Gannon, and R.J. Douglass, The Characteristics of Parallel Algorithms, MIT Press, Cambridge, MA 1987.
29. N. Karengelen, "Multi-Domain Real-Time System Design, Capture and Analysis", Proc. 1st System Design Synthesis Technology Workshop, September 1991.
30. S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi, "Optimization by Simulated Annealing", Science, May 13, 1983, Volume 220, No. 4598.
31. P. Le Guernic, T. Le Guernic, M. LeBorgne, and C. Le Maine, "Programming Real-Time Applications with SIGNAL," Proc of IEEE, Vol. 79, No. 9, pp. 1321-1336/
32. I. Lee, and S. Davidson, "The CSR Paradigm for Real-Time System Specification and Analysis," DAPRA Software Technology Conference, 1992, Los Angeles, CA, April 1992.
33. E. Lock, N. Prywes, and S. Andrews, "Case For Development And Re-engineering Of Real-time Distributed Applications", Fourth International Conference, Software Engineering and Its Applications, Toulouse, France, December 9-13, 1991.
34. S. Lee, and J. K. Aggarwal "A Mapping Strategy For Parallel Processing", IEEE Transactions On Computers, Vol. C-36, No. 4, April, 1987.
35. E. Lock and S. Andrews, "Systems Engineering Methods and Automation for Development and Enhancement of Integrated Multi-Component Military Systems", Proc. 1st Systems Design Synthesis Technology Workshop, September, 1991.
36. T. A. Ly, J. T. Mowchenko, "Applying Simulated Evolution To Scheduling In High Level Synthesis", Proc. 33rd IEEE Midwest Symposium On Circuits And Systems, August 1990.
37. Z. Manna, A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems: Specifications, Springer, New York, 1991
38. J.J. Molini, S.K. Misra and P.H. Watson, "Real-Time System Scenarios", Proc. 11th Real-Time Systems Symposium, IEEE Computer Society Press, Los Alamitos, CA, December 1990, pp 214-225.
39. C. Nguyen, S. Howell, P. Hwang, Systems Design Factors, NAVSWC TR-92-XXX, February 1992.
40. J. Pearl, Heuristics: Intelligent Search Strategies For Computer Problem Solving, Addison-Wesley Publishing Company, Inc., Reading, MA, 1984.
41. D. Peng, and K. Shin, "Modeling Of Concurrent Task Execution In A Distributed System For Real Time Control", IEEE Transactions On Computers, Vol. C-36, No. 4, April 1987.
42. N. Prywes, E. Lock, and X. Ge, "Automatic Abstraction of Real-Time Software and Re-Implementation in Ada", Proc of Tri-Ada '91, October 21-25, 1991.

43. A. Round "The Evolution of Knowledge-based Simulation", in "The Handbook of Artificial Intelligence", Volume IV, Barr, Cohen, Feigenbaum, ed. Addison-Wesley 1989.
44. G. Staples, D. Shavon, "Earthquake Insurance: One Integration Approach", Software Magazine, pp 41-44, February 1992.
45. E. G. Sable, P.B. Shanghavi, G.Y. Wyatt, eds. "Intelligent Networking - Network Systems," Special Issue AT&T Technical Journal, Summer 1991, Vol. 70, No. 3-4
46. R.W. Selby, A.A. Porter, D.C. Schmidt, J. Barney, "Metric Driven Analysis and Feedback Systems for Enabling Emperically Guided Software Development," Proc. 13th International Conference on Software Engineering, 1991
47. L. Sha, J. P. Lehoczky, R. Rajkumar, "Task Scheduling in Distributed Real-Time Systems," Proceedings of 1987 IEEE IECON, P. 909-916.
48. Y. Shi, N. Prywes, "Generating Multitasking Ada Programs from High-Level Specification," IEEE Third International Conference on Ada Applications and Environments, pp. 137-149, Manchester, New Hampshire, May 23-25, 1988.
49. Y. Shi, "Very High Level Concurrent Programming," IEEE Transactions on Software Engineering, Vol. SE-13, No. 9, September 1987.
50. P. Ward, and S. Mellor, Structured Design, Yourdon Press, Englewood Cliffs, NJ, 1972, 2nd ed
51. G. Wiederhold, "Model-Free Optimization," DARPA Software Technology Conference, 1992, pp. 83-96.
52. J.C. Wileden, A.L. Wolf, W.R. Rosenblatt and P.L. Tarr. "Specification Level Interoperability," CACM, 34(5) May 1991

Benefits of Using Object-Oriented Methodology for Missile Guidance Processor Software Development

Bruce Hoover
Graduate Studies
Governors State University
University Park, IL 60466

Sungyoung Lee
College of Arts and Sciences
Governors State University
University Park, IL 60466

ABSTRACT

The purpose of this paper is to epitomize the benefits of using the object-oriented methodology to specify and develop a real-time system. Object oriented is a methodology by which abstract implementation-oriented classes are developed. Classes denote groups of system objects that share common characteristics and behavior. These classes are further specified by mapping data "attributes" and procedural "methods" to particular class "instances". Finally, the class interfaces or "message" structures are defined. It has been discovered that the object-oriented approach can reduce development time of software systems [1]. This paper will show the object-oriented

methodology in specifying an existing case study [3]. This case study is of a real-time software system designated the VGPS which implements a missile guidance processor control system.

INTRODUCTION

In this paper the main system to be examined is the Operational Flight Program (OFP) [3]. The OFP is the single system user that has three primary processes: 1) Autopilot, 2) Guidance, and 3) Gain Computer. The Operational Flight Program interfaces are identified in Figure 1. The following are commonly used abbreviations [3]:

FCS: FIRE CONTROL SYSTEM

T/M: TELEMETRY

IRU: INERTIAL REFERENCE
UNIT

R/T: RECEIVER / TRANSPONDER

CAS: CONTROL ACTUATION
SYSTEM

S/A: SAFE AND ARM

GPU: GUIDANCE PROCESSOR
UNIT

P/F: PROXIMITY FUZE

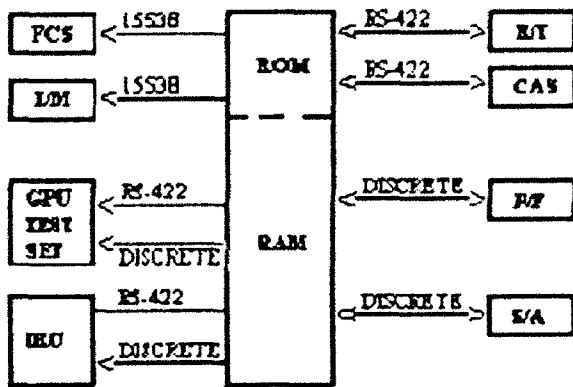


Figure 1 - VGPS INTERFACE BLOCK DIAGRAM

In Object-Oriented (OO) methodology a system is a group of objects that communicate among themselves. This communication can be understood by analyzing the relationships that exist between objects, the behavior of each individual object, and the mutual behavior of cooperating objects.

Objects themselves may also be considered systems. For benefit of this case study, the OFP is a system of smaller objects that interact to produce the behavior associated with the object "OFP". An OO specification attempts to modularize a system along the same object boundaries that exist within the real-world. This leads to a one-to-one correspondence between the object as it exists in the real-world and the object as it exists in a specification. Herein lie one of the principal benefits to be realized using an object-oriented specification. In addition to simplifying the conceptual model of an object's specification it also serves to organize all knowledge about each object in a single logical location.

Because of the emphasis on objects and the corresponding de-emphasis on processes, the OO approach to system specification encourages the specifying of logical systems rather than physical systems. The necessary functional properties of a system may easily be expressed, but these take priority over the specification of system

objects and their relationships and interactions. OO techniques also provide forms of conceptual abstraction not found in other techniques that rely heavily on the physical approach.

In addition to aggregation it provides generalizations and classification. These allow it to easier specify the informational processing with respect to objects. Most importantly the object methodology implicitly leads to the construction of systems that embody the five attributes of well-structured complex systems:

- 1) Complexity takes the form of a hierarchy where every complex system is composed of related subsystems who themselves are composed of subsystems until the elementary components are reached [2].
- 2) The choice of what constitutes an elementary component is arbitrary and is totally dependent on the observer of the system,

- 3) "Intracomponent linkages are generally stronger than intercomponent linkages. This fact has the effect of separating the high-frequency dynamics of the components - involving the internal structure of the components - from the low-frequency dynamics - involving interaction among components" [2] this leads to a separation of concerns allowing the observation of various parts of the system to be encapsulated.
- 4) "Hierarchic systems are usually composed of only a few different kinds of subsystems in various combinations and arrangements" [2].
- 5) "A complex system that works is invariably found to have evolved from a simple system that worked.." [5] that is to say that as systems evolve, objects that were once considered complex now become the elementary objects upon which more complex systems are constructed.

The following benefits of applying the object methodology have been discovered:

- (1) Lower development risk.
- (2) Allows systems to be developed that are more adaptable.
- (3) Modern software practices are implicitly involved.
- (4) Allows system development to be more natural in it's approach.

LOWER DEVELOPMENT RISK

The risk of developing systems with the object methodology is lowered for 3 reason(s). First, integration is spread out across the entire system life cycle. Second, during the object methodology design phase an intelligent separation of concerns reduces development risk and also inherits the benefits of incremental development. Lastly, due to separation of concerns a complex system can be readily identified for its correctness. In this case study a real-time system was needed to be developed in a timely manner that would be totally operable with little maintenance.

The distribution of system integration and separation of concerns has the benefit of minimizing the impact of inevitable system

changes. The object classes developed in earlier phases of the system life cycle allow the introduction of changes lower in the hierarchy to be made with no effect to the higher level objects. This level of abstraction is a natural benefit of developing objects from inheriting methods from subclasses.

As in the original case study [3] the careful design of object classes allowed the full advantage of incremental development to be realized. This allows early functionality and evaluation of system capability which in the development of a complex hierarchical system leads to a reduction in the time needed for integration, and allows system evaluation to occur at an earlier phase of system development. A side benefit of early system evaluation is a capital savings of several magnitudes as it is well known that the earlier a system modification is made in the system development life cycle the less costly that modification is to administer.

ADAPTABILITY

In approaching the topic of adaptability it must be remembered that several layers of abstraction were used in the original case study design to facilitate this feature. Therefore it can be seen that by providing "sufficient" level of abstraction a system can be designed to be adaptable in the sense that the overall functionality will not change drastically (i.e If that was the case then design a new system) therefore the essential system will be stable and the "how" to implement this essential system will change to a measurable degree. A "properly" designed OO system will exhibit adaptability in that it's superclasses will be affected very minimally due to changes at the base class level. In the cases where this is not immediately apparent then another level of abstraction need be added to insulate the superclasses from the base classes. OO methodology supports adaptability through the mechanisms of abstraction, encapsulation, modularity, and hierarchy.

Abstraction is one of the principles that we use to cope with complexity. Hoare

suggests that "abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon these similarities and to ignore for the time being the differences" [6]. Through the development of objects (Hardware/ Software) from base classes that were analyzed for correctness and functionality the VGPS system implementation utilized information hiding/method hiding. This allows the higher level objects to be "insulated" from the changes made to lower level objects. The advantage to system development is that now unimplemented objects can be readily accessed by using stub functionality.

Encapsulation is the principle by which no part of a complex system should be dependent on the internal implementation of another part of the system. Whereas abstraction helps one conceptually model a system to the desired level of focus encapsulation "allows program changes to be reliably made with limited effort" [7]. In the initial class designs for the VGPS it

was decided that the elementary hardware dependent components should be encapsulated to allow the higher level classes to function relatively independent of the underlying hardware system. Similar to the original case study this also had the benefit of allowing a consistent interface to several system support objects. This allowed the design and construction of high level objects that did not need modification as hardware changes were instituted and the fully operational Operational Test Program implemented in final form.

Modularity is the principle by which a system is partitioned into individual well-defined components. This allows easier comprehension and reduces the complexity of a system. Modules serve as physical containers in which we implement the classes and objects of our logical design. Therefore it can be seen that the principles of abstraction, encapsulation, and modularity are cooperative in their functionality as far as the object methodology is concerned. That is an object provides a well-defined boundary for

a single abstraction, and encapsulation and modularity provide the necessary barriers around the object. The application of modularity in the VGPS system was the natural result of designing meaningful base classes. The principles of encapsulation and separation of concerns that were utilized in the base class design for the construction of meaningful base classes naturally lead to the development of modular software. Furthermore the principles of encapsulation and separation of concerns lead to increased cohesion and a decreased coupling.

Hierarchy is the "...ranking or ordering of abstractions" [4]. This was accomplished in the VGPS by the successive use of base classes to form objects which in turn were the base "classes" for more complex objects. The hierarchy allowed the development of a system that was hardware independent and very adaptable. The hardware independence was realized through the careful design of the interface between hardware dependent base classes and their respective superclasses.

Superclasses represent a more general abstraction and therefore as objects were built with these superclasses those objects became dependent on the functionality of the superclasses themselves and not the underlying implementation. Subsequently this would allow the complete development of those objects that interfaced with a particular superclass even though the procedural methods of that superclass were potentially volatile.

MODERN SOFTWARE TECHNIQUES

The modern software engineering techniques that were used in the construction of the VGPS as in the original case study include information hiding, separation of concerns, modularity, adaptability and incremental development. Again as stated in the original case study the application of these techniques has not been studied extensively in real-time systems development therefore it is not our theoretical claim to identify the advantages and disadvantages of their use in system development.

However, because the object methodology supports most of these techniques implicitly it can be shown that the overall construction of a system using this methodology will as a rule take less time to develop and maintain. This is primarily true because the object methodology stresses the essential specification of a system during the base class design phase. The formation of superclasses from base classes further facilitates the development of systems that are stable and correct because the designer can comprehend more of the system's functionality and therefore can at once see the correctness or impropriety of a particular relationship or interface. Furthermore, the resulting class re-use from a stable working system (i.e. which is a byproduct of the object methodology) leads to the formation of even more complex systems that are correct and stable with little effort. Maintaining and enhancing systems developed with the object methodology tend to be easier due to the incorporation of separation of concerns, encapsulation, and modularity. That is a particular enhancement can be made to a

advantages garnered from designing/ implementing via the object methodology versus designing/ implementing with a methodology and adding in these "extras". As stated before the use of these modern software techniques have not been fully documented as far as the respective benefits and advantages to be realized from their use. But it is not a far leap of faith to justify that a system designed/implemented with these tools will allow the system designer to faster implement systems that are more reliable.

REFERENCES

- [1] Schmucker, K. "Object-Oriented Programming for the Macintosh", Hasbrouk Heights, NJ:Hayden 1986, p. 11.
- [2] Simon, H. "The Sciences of the Artificial", Cambridge, MA: The MIT Press 1982, p. 218.
- [3] Bond, B., Bemrich, S., Connelly, J., Pendergrass, G., and Hulsey, J., "Missile Guidance Processor Software Development: A Case Study", IEEE Real-Time Systems Symposium, December, 1988, pp. 60-68.
- [4] Druke, M., Quoted From "Object-Oriented design with Applications", Benjamin/Cummings, 1991, pp. 188.
- [5] Gall, J. "Systemantics:How Systems Really Work and How They Fail", 2nd ed. Ann Arbor, MI: The General Systemantics Press 1986, p. 65.
- [6] Dahl, O., Dijkstra, E., and Hoare, C. A. R. "Structured Programming", London England: Academic Press, 1972 p. 83.
- [7] Gannon, J., Hamlet, R., and Mills, H. "Theory of Modules", IEEE Transactions on Software Engineering vol SE-13(7), 1987 p. 820.

EMBEDDED COMPUTER SYSTEM REQUIREMENTS METHODS ANALYSIS AND IMPROVEMENT

Stephanie M. White
Grumman Corporate Research Center
Mail Stop A08/35
Bethpage, New York, 11714
(516) 575-2201

Abstract: Requirements methods proven practical on large embedded computer systems (ECS) are formalized, synthesized, and improved. A cross-section of methods are evaluated for robust semantics, mathematical foundation, capability for analysis and verification, and support for model construction, comprehension, reuse, and modification. A synthesized Pragmatic Formal Method (PFM) augments the best characteristics of current methods. Some of the capabilities defined for PFM could be added to current methods, supporting analysis for errors of omission and commission, and facilitating system simulation and early prototyping.

ECS ENGINEERING

An embedded computer system (ECS) is part of a physical system and must perform quickly and correctly for the physical system to perform its intended function. These systems interact with their environment (people, hardware devices, and software in other systems), and are composed of subsystems that may themselves be composed of smaller subsystems interacting with each other and their environment, see Fig. 1. Each subsystem usually contains both hardware devices and software. The hardware devices may be generic processors or displays, or may be functionally specialized for an application, (e.g., radars, communication, and navigation devices). Each hardware device may also contain software. The software and hardware interact with each other and with the environment.

Systems Engineering: For embedded systems, system requirements and design must be performed before software requirements can be specified. Systems engineering is engineering from a total system, rather than from a component viewpoint. The system engineer analyzes end-to-end critical processing threads, including analog parts that affect performance and accuracy, to develop a cost-effective, feasible design that meets mission requirements. He/she allocates system requirements to hardware, software, firmware, communication links, and people; and then allocates software requirements to distributed subsystems and distributed databases. A systems engineer must have a strong software engineering

background to make decisions that avoid software development problems.

Systems that process and control information are usually complex; many of these systems are unprecedented and requirements change frequently. Discipline, clarity, and automation are needed to design such systems. System engineers should be concerned with three fundamental concepts (TRUX 72)

- Modeling - a quantitative description of system operation.
- Dynamics - the change in the system with respect to time.
- Optimization - choice of a good design based on relative weights assigned to system aspects.

Requirements Models: ECS are primarily control oriented and require different techniques for system development than transaction-based systems. Embedded systems must respond quickly and correctly to complex sequences of unpredictable external events. The system response varies according to the order of these events. Consequently, the accurate description and analysis of the desired system reaction to sequences of external stimuli is a very important step in the specification of embedded systems. The model that embodies hierarchy and dataflow, which is sufficient for transaction-based systems, is insufficient for real-time systems. Requirements must include the dynamic view that specifies changes in system state caused by external events.

The boundaries of the system must also be explicitly defined, and the environment of the system must be modeled and analyzed as part of the system requirements development process. The model should include the time-dependent, deterministic and non-deterministic, parallel and serial nature of the inputs.

Nonfunctional requirements such as performance, fault detection and recovery, safety, security, availability, reliability, and ease of change are major concerns in complex embedded computer systems. They must be addressed during the generation of system and software requirements.

Mathematical Formalisms: ECS are complex and a single mathematical formalism is insufficient to define all system aspects. It is necessary to find a set of formalisms that provides a minimal cover of those

aspects necessary for system description. Quantitative models based on mathematical formalisms are needed to support tradeoff analysis, and the development of quality designs. Mathematical formalisms are required because of the size of the problem. Mathematics provides the discipline, rigor, and reasoning ability that are needed to solve complex problems.

METHOD EVALUATION

Grumman has sponsored research in requirements definition methods from the late 70's through the present (BARI 79, WHIT 81, WHIT 85a, WHIT 85 b, WHIT 86, WHIT 87a, WHIT 87b, WHIT 92a, WHIT 92b). Grumman also performed research in requirements definition under contract to Naval Air Development Center (WHIT 80), and Naval Research Laboratory (WHIT 83, NASH 84, KMIE 84).

Methods Evaluated: In (WHIT 87b), we examined eight methods for modeling reactive systems to determine the formalisms used, and to evaluate method capability for supporting embedded system specification. These methods have not changed appreciably since 1987. New object-oriented analysis methods (COAD 91, SHLA 88) use an entity-relationship approach to information modeling. These methods provide an important view, but are not complete. They do not address the scenario or control aspects of ECS requirements.

The methods we examined are the semi-formal/formal methods most commonly used in industry for modeling complex systems. They are:

- Distributed Computing Design System, a function flow and dataflow method
- Structured Analysis for Real-Time Systems, (SA-RT), a dataflow and state-machine method
- Higher Order Software (HOS), a function composition method
- Jackson System Development (JSD), a data (event) structure method
- Software Cost Reduction (SCR), a method based on cooperating sequential processes and event-action causality
- PAISLEY, a method based on cooperating sequential processes and functional programming
- STATEMATE, a method based on cooperating sequential processes and state decomposition
- PAMELA, an object-oriented requirements analysis method.

The mathematical basis of these methods are: input to output mapping, algebra (data abstraction), process abstraction, function composition/ decomposition, sequential machines, concurrent machines (cooperating sequential processes), extended abstract machines¹, and predicate logic.

Techniques for Method Evaluation: The following techniques were used to compare methods:

- Each method was used to specify the requirements for an embedded system. A home heating system was used for benchmark comparison. Based on this exercise and the literature, we evaluated each method with respect to thirteen desirable method characteristics.

- We compared each method's model with a generic model. A formal Entity-Relationship (ER) model (CHEN 76) was used to compare the expressive power of methods. Expressive power is important as statements that cannot be expressed in a model will be omitted from the analysis and resulting specification.

- Methods were analyzed to determine whether they use a partial order temporal approach, specifying maximal concurrency and nondeterminism. All methods specify some concurrency but most only specify deterministic flow. Within processes, most methods use a linear or branching order concept. Linear methods are the least powerful and partial order approaches are the most powerful (PNUE 86).

Formal Characterization of Method Models:

Experimentation with method models has shown that they can all be considered as special cases of the ER model (TEIC 80).

Formalizing system definition methods using ER models has several advantages:

- The method objects and relationships are precisely defined.
- The requirements knowledge captured by the methods can be stored in a database for analysis and retrieval.
- The expressive power of method languages can be compared.

Entity-Relationship Representation of Generic Real-Time Model:

The Entity-Relationship model in Fig. 2 is used as a baseline to analyze a method's underlying model. The generic model is a simplified one, but includes environment as well as system functions, which illustrates whether the method addresses environment modeling. The generic model also decomposes state, function, event, and data. This is used to demonstrate that the evaluated methods decompose either event or state, but not both (e.g., JSD decomposes events while STATEMATE decomposes state). Decomposition of both state and event would provide more power for tracing system requirements to software requirements.

¹An extended abstract machine is an abstract state machine that permits condition "guards" to be associated with events. This makes the specification more succinct as the machine has knowledge about states of other machines, and every event sequence does not have to be specified.

In the illustration, rectangles represent objects; circular nodes represent relationships; the nodes are numbered for identification. The relationship holds in both directions; the relationship name has been provided for only one direction to keep the diagram simple. For example, relationship (1) is read ENVIRONMENT contains FUNCTION, but the complementary relationship FUNCTION contained in ENVIRONMENT also holds.

In the generic model, both the ENVIRONMENT and SYSTEM contain FUNCTIONS. A FUNCTION is performed in certain STATES, causes and is triggered by EVENTS, is a refinement of a more abstract FUNCTION and uses and sets DATA. STATES can have subparts, and are contained in a STATE-MACHINE-ABSTRACTION, which can be a refinement (detailed definition) of a more abstract STATE. EVENTS contain and affect STATES. EVENTS are decomposed, and the levels of event refinement relate to the levels of data refinement. EVENTS contain EVENTS, and an EVENT at an abstract level is started and ended by EVENTS at a more detailed level. EVENTS are a function of DATA, which is also decomposed.

METHODS EVALUATED

Four methods, DCDS, SA-RT, SCR, and STATEMATE are described in detail. A set of tables compares the eight methods identified above. The eight methods are described in more detail in (WHIT 87b).

DISTRIBUTED COMPUTING DESIGN SYSTEM (DCDS): DCDS was developed as a real-time system methodology at TRW, based on research sponsored by the U.S. Army Ballistic Missile Defense Advanced Technology Center, BMDATC (ALFO 85). Research began in 1973. The method includes a System Requirements Engineering Methodology (SYSREM), and a Software Requirements Engineering Methodology (SREM). SYSREM is used for defining system requirements and allocating them to the data processing subsystem and the hardware. SREM is then used for defining the software requirements.

System Requirements Model: An important concept of SYSREM is that a system function acts over a finite interval of time. This time interval is the maximum computation time and is specified as a performance requirement. A system action is represented as a function that accepts items arriving during the time interval and transforms them into items that appear as output during the time interval.

The system requirements model is based on a stimulus-response graph model, called an F-net, with nodes representing subfunctions and edges representing events, or flow due to function completion. The system

accepts a sequence of inputs and produces a sequence of outputs. In applying the method, normal functionality is modeled first, then exception conditions are added. At the primitive level, the graph model is equivalent to a set of state machines which receive inputs and produce outputs. Some of these state machines are realized by hardware and some by software. The software is allocated to processors and interface designs are derived.

Software Requirements Model: After system requirements are allocated to processors, SREM is used to define data processing requirements. For each processor, a set of graph models called R-nets define paths from input to output interfaces. Each R-net has a single entry, which may be an interface to the environment, and one or more exits, which are either interfaces to the environment or terminators. Processor inputs and outputs arrive and depart at ports, called interfaces, and are identified as messages from a hardware device or another processor. R-nets are composed of subnets, which are similar to R-nets and which can be further decomposed. At the primitive level, nondecomposable functions, called alphas, use inputs and data derived by other alphas to determine the value of new data items. An R-net can trigger R-nets via an event. A delay associated with the event causes the event to take place at a later time. Validation points are nodes which can be added to an R-net for specifying a path of processing through the net and its subnets. Data is recorded at each validation point during dynamic specification traversal, simulating system operation. Minimum and maximum response times can be expressed for paths, and accuracy requirements can be expressed for data.

Evaluation of DCDS with Respect to Generic Model: Both the software system and its environment (the hardware) contain functions. DCDS describes functions, using both a function and net concept. The net details the abstract function and consists of subfunctions. Functions cause and are triggered by events, and use and set data. At the primitive level in SYSREM, the nets are equivalent to state machines and function is synonymous with state. States and events cannot be decomposed. Relationships in the generic model which are not expressible in DCDS are illustrated in Fig. 3. Rectangles denote objects and circles denote relationships between these objects. The relationship is identified by the number in the circle. Relationships 3,5,6,7,10,11,12 are not expressible in DCDS. This is illustrated by the lack of shading in the figure.

DCDS Temporal Approach: DCDS can specify concurrency and selection, but not nondeterminism. Within SREM, maximal concurrency is not specifiable and is limited to fan-out, fan-in along a path. In some

cases the limitations of linear temporal order are imposed. The modeler is forced to show one path. Alternate design paths cannot be shown.

YOURDON STRUCTURED ANALYSIS REAL-TIME (SA-RT): The Yourdon SA-RT methods are primarily based on dataflow (WARD 85). SA-RT modeling begins with the development of an external event list, to help engineers focus on those events in the environment to which the system must respond. The events, which are labeled as data or control, are used to define the system boundary and first level decomposition.

SA-RT Model: The model contains a hierarchy of data flow diagrams (DFDs). The top level, called the context diagram, describes the interface between the system and its environment. It consists of one data transformation which defines the system, and dataflows which define the system inputs and outputs. Lower level DFDs contain data transformations which perform the system function and which pass data (dataflows) among themselves. Data can also pass between DFDs at the same level. Data which is not immediately used is retained in "datastores". Primitive (not decomposed) data transformations are described in minispecs, which can be written in structured English or a specific program design language (PDL), or can be described by decision tables. Data structure is defined in a data dictionary.

A control transformation is detailed by a diagram called a state transition diagram (STD), similar in style to a Mealy sequential machine model. The STD activates and deactivates processes based on the history of signals from the external environment and other processes, and sends signals to other control processes and the environment. When events are primarily data oriented, the DFDs are drawn first; when events are primarily control oriented, so is the system, and the STDs are drawn first.

SA-RT Temporal Approach: SA-RT specifies concurrency by modeling concurrent STDs and by activating parallel processes. Maximal concurrency could be specified but that is unlikely, as maximal concurrency is not a primary concern of SA-RT. The SA-RT dataflow concept forces the SA-RT modeler to show one path; design alternatives cannot be shown.

Evaluation of Yourdon SA-RT With Respect To Generic Model: SA-RT models EVENTS in the ENVIRONMENT but not FUNCTIONS and thus cannot express relation (1) in the generic model. Events are not decomposed so SA-RT does not express relations (7,10,11,12), as shown by the lack of shading in Fig. 4, SA-RT Compared to Generic Model.

SOFTWARE COST REDUCTION (SCR): The Software Cost Reduction (SCR) project was established by Naval Research Laboratory (NRL) to prove that modern software engineering practices could be used for large Navy Software projects. The A-7E Aircraft software was rebuilt as a test.

Support for Information Hiding: The SCR Software Requirements Specification is based on the principles of separation of concerns and information-hiding (HENI 78, HENI 80, WHIT 83). If hardware dependent information is needed, the analyst looks at the Input and Output Data Items section of the SRS. This section is organized by hardware device, and by input and output name. The information in this section is not repeated elsewhere in the SCR document. Only the acronyms for input and output names are used. If hardware related attributes change, only the Input and Output Data Items section should have to be changed.

The software functional requirements are described by Modes of Operation, and by Time-independent Description of Software Functions. If the definition of environmental events and required software actions change, only these sections should have to be changed. Timing requirements are specified by function and are listed separately. If the timing requirements change, only this section should have to be changed.

The SCR document has a placeholder for a section on accuracy requirements. A format for these requirements is not defined. The *Undesired Event Responses* section contains a list of events which could occur and what response is desirable.

The SCR Required Subsets section identifies subsets of services which could be useful, if isolated, in the development of similar systems. The Expected Types of Changes section identifies areas of possible future change so that the design can accommodate these.

The SCR Glossary provides a definition of abbreviations, acronyms, and technical terms. Sources for information are given. Alphabetical Indices provide access to inputs, outputs, modes, and functions. The SCR dictionary contains text macros. Text macros are phrases used to simplify the document.

SCR Model: Templates are completed for inputs and outputs. A function is defined for each output. Functions are differentiated as either periodic (synchronous) or demand (asynchronous). A periodic function may be initiated and terminated or may always operate. Tables are used to define events and conditions that cause a change in output value or that activate/deactivate periodic functions.

Because functions differ greatly in different modes, modes are used to simplify the function description. A mode in the SCR methodology is a system state defined by the history of events in the system. The modes are grouped by mode class. The system can be in more

than one mode class at a time. The modes within each mode class are mutually exclusive.

Text macros take the place of compound conditions and data item definitions, when the data items are not outputs or directly derived from inputs. The use of text macros supports information hiding. If there is a change in the set of compound conditions or in the data item definition, only the dictionary has to be changed, not the entire document.

Evaluation of SCR Methods With Respect to Generic Model: The SCR methods do not model functions in the ENVIRONMENT, so relation (1) in the generic model is not expressed (see Fig. 5). STATES and EVENTS are not decomposed, so state decomposition relations (5,6) in the generic model are not expressed. Events are not decomposed, so relations (7,10,11,12) are not expressed.

SCR Temporal Approach: The SCR methods are based on a partial order concept. Since the input to output transformation is not identified, both nondeterminism and maximal concurrency can be specified. The extended machine concept simplifies the definition of required temporal order.

STATEMATE: The STATEMATE model is a graphic model based on cooperating sequential processes, extended abstract state machines, predicate logic and dataflow (ADCA 85, HARE 86).

STATEMATE Model: The model contains templates, Statecharts, and Activity Diagrams. Templates are completed for the following objects: state, condition, event, action, activity (function), signals/variables, modules, and channels (which connects modules). A Statechart is a visual extension to conventional State Transition Diagrams used in SA-RT. An Activity Diagram shows data and control flow and is similar to a dataflow diagram in SA-RT.

The Statechart is a major contribution of STATEMATE. A number of state transition diagrams can be shown on the same chart, displaying parallelism, selection, and decomposition. STATEMATE can model cooperating sequential processes or support structured analysis methods.

Evaluation of STATEMATE With Respect to Generic Model: STATEMATE does not decompose EVENTS. STATEMATE can express all relations in the generic model except event decomposition relations (7,10,11,12), as shown in Fig. 6.

STATEMATE Temporal Approach: If Statecharts were used alone, STATEMATE would be based on a partial order concept. With the incorporation of user specified Activity Diagrams, a transformation from input to output is defined and STATEMATE is reduced to a

modified branching logic. Within the branching logic, maximal concurrency and nondeterminism cannot be specified.

COMPARISON OF METHODS WITH RESPECT TO METHOD ATTRIBUTES

The evaluation of eight methods is summarized in table 1. Methods are judged with respect to thirteen desirable method characteristics. To clarify usage of each method's vocabulary in the remainder of the table, the first row in sheet 1 of the table compares method objects to a set of generic objects. The last two rows in sheet 3 identify format and automated support for each method.

Table 2 compares the eight methods by assigning a value (0, 2.5, or 5) which measures the capability of each method with respect to each attribute. A value of zero indicates the method is poor. A value of 2.5 indicates the method is fair. A value of five indicates the method addresses the attribute well. The SCR methods score the best with a total score of 47.5, which shows these methods to be far superior. DCDS follows with a score of 30. STATEMATE received the third highest score, 25.

Formal Basis (Attribute 1): Current methods are based on the following formal concepts: (1) input to output mapping, (2) function composition/decomposition, (3) process abstraction, (4) data abstraction, (5) cooperating abstract machines, (5⁺) cooperating extended abstract machines, and (6) predicate logic.

- DCDS is based on (1), (2), (3), (5)
- HOS is based on (1), (2), (3), (4)
- JSD is based on (1), (2), (3)
- SA-RT is based on (1), (2), (3), (5)
- SCR is based on (1), (2), (3), (5⁺), (6)
- STATEMATE is based on (1), (2), (3), (5⁺), (6)
- PAISLey is based on (1), (2), (3), (5)
- PAMELA is based on (1), (2), (3), (4)

SCR and STATEMATE are the most formal of the evaluated methods.

Model Construction (Attribute 2): The SCR model is the easiest to construct and change, as long as mode definitions are stable, as each data item, function producing output data, and text macro (definition of compound conditions or function producing internal data) is defined in one and only one place, and there are no explicit interconnections between functions. A problem would arise if there were major changes to mode definitions, as function definition is dependent on modes. If utilizing SCR methods for model construction, modes should be introduced after requirements have stabilized.

Model Comprehension (Attribute 3): The SA-RT method provides a good visual overview of a logical design. Requirements are clearest in the SCR model as they are explicitly defined in event-action causal statements: "If event a occurs and condition b is true, perform action c". There is a problem in understanding how the model works together as this is not explicitly defined. There is no way to tell if all the requirements are present or if the system will work, if built to these requirements. The static interrelationship of SCR model parts could be shown using diagrams similar to dataflow diagrams. These diagrams should be generated from the SCR model. If methods defined by Grumman were automated, the SCR model would be executable. Then, the dynamic interrelationship of parts could be shown by running the model.

Design Independence (Attribute 4): If modes were not used, the SCR model would be design independent; the modes impose design constraints. All other method models are design dependent. They provide a design which is independent of physical constraints.

Stepwise Refinement (Attribute 5): No model adequately refines state, function, event, and data, and relates them so that there are well defined functional layers. STATEMATE has developed the best definition of state decomposition, but engineers using the method find it difficult to relate decomposition of state to decomposition of function. These engineers use structured analysis methods and turn processes on and off by a state transition "controller" at each level. Our research extends the SCR methods by expanding requirements defined in "text macros." When text macros define conditions not directly derivable from inputs, a function is needed to evaluate the condition. These functions become additional cooperating processes. In addition, we decompose state, function, event, and data so that system requirements can be traced to software requirements, and levels of abstraction can be defined.

Separation of Concerns (Attribute 6): The SCR methods are the only methods that have made separation of concerns a goal and provide the best support. All other methods show explicit interconnections between model parts, and would be difficult to partition according to the concerns of different specification readers.

Nonfunctional Requirements (Attribute 7): It has been shown (WHIT 83) that functional timing constraints could differ by mode; and it is therefore reasonable to believe they could differ within a function according to event, condition, or output value. The SCR methods address timing constraints well, but they can only be associated with data or function. PAISLey addresses timing constraints, but also associates them

with function or process. DCDS associates timing constraints with objects (data, events, functions) or stimulus response path, and thus is the most powerful in this regard. Other nonfunctional requirements are not as well addressed as timing. PAISLey specifies function reliability. DCDS associates nonfunctional requirements with all objects, but the requirements are specified textually.

Design Tradeoff Analysis (Attribute 8): Design tradeoffs for distributed processing depend on external event frequency, periodic and aperiodic inter-system message flow, nonfunctional requirements, estimated data and function size, scheduling constraints, communication protocols, and on processor, storage, and communication device constraints. Depending on allocation, fault detection and analysis procedures vary and must be considered. Tools are available (FRAN 87) for performing analysis, but the evaluated methods do not provide much of the data needed for using this type of tool.

Test Identification (Attribute 9): A product acceptance specification defines externally visible behavior which the product must demonstrate, and specifies any design constraints that must be met. The SCR methods provide the most support for identifying acceptance tests, as they describe behavior which is externally visible or which could be externally determined. STATEMATE Statecharts support the identification of tests as the flow of events and actions are identified. DCDS functional paths support test identification, but it is difficult to determine which paths should be tested other than the ones shown. Other methods do not support testing well.

Identification of Missing Requirements To Be Supplied TBS(s) (Attribute 10): The SCR templates and the use of text macros for information hiding provide the best support for identification of TBS(s) or gaps. It is harder to support this feature with a method which is based on flow.

Verification (Attribute 11): An entity-relationship (ER) model and operational capability are desirable for supporting model analysis and verification. PAISLey, STATEMATE, and DCDS provide the best facilities for model verification. All three methods provide an operational capability. A PAISLey definition is statically analyzed in a manner similar to a software program. STATEMATE and DCDS perform static verification based on the underlying ER model; the database is queried for information. We have defined an ER model that would support SCR model capture and dynamic execution. Our decision tables, predicate logic, and temporal logic support static verification.

Reusability (Attribute 12): The SCR methods identify subsets of the software which could be replaced or be reusable. PAMELA packages data and function by object in Ada PDL, and these packages may be reusable. HOS abstract data types and JSD object structures may be reusable. STATEMATE Statecharts identify control. They may be reusable, if they are organized according to objects in the environment. Our method packages requirements by object, to provide a flexible reusable specification.

System and Software Definition (Attribute 13): Only DCDS supports both system and software definition and provides traceability from one to the other. Unfortunately, the path concept which is used in DCDS requires that design decisions be made when developing the model.

Comparison Of Methods With Respect To Generic Real-Time Model: None of the evaluated methods contains all objects and relationships in the generic real-time model. JSD is the only evaluated method that decomposes EVENT, but does not decompose STATE or STATE-MACHINE. SA-RT and PAISLey have STATE decomposition. The SCR method uses STATES and STATE-MACHINES, but does not decompose them.

Comparison Of Methods With Respect To Temporal Approach: The most powerful partial order methods are those that (1) can express maximal concurrency, (2) specify mutual exclusion at every level, and (3) specify nondeterminism without defining all possible paths. The SCR methods support all three. Item (3) is supported through the use of extended machines. This is preferable to defining all possible paths which makes the definition complex. The paths can be generated from the extended machine, if desired.

A PRAGMATIC FORMAL METHOD (PFM)

Our evaluation of practical methods has identified the Software Cost Reduction method (SCR) as having the best characteristics on which to build a more powerful method. SCR methods have a strong formal basis and can be augmented to eliminate deficiencies. We are developing a method, PFM, that enhances the SCR methods with capabilities found to be beneficial in other methods, and features not available in any evaluated method. PFM is discussed in detail in (WHIT 87b). PFM logic analysis techniques have been helpful in checking aircraft mode transition. Other capabilities have not as yet been tested.

Enhancements to the SCR methods include:

- (1) A formal Entity-Relationship model for the SCR method. This model is a formal description of the

information captured about a target system when using the SCR method. (Examples of entities are modes, events, and actions; examples of relationships are caused-when, makes-true, and activated-by.)

- (2) A mapping from the static SCR model to a net for behavioral execution. PFM-nets are similar to Petri nets, and are not dependent on textual context for execution. They should execute at the speed of Petri-nets.

- (3) Support for system and software definition. Early in system definition, system engineers understand conditions that affect the system, but have not as yet decided on system modes. Early system models should be developed without modes, and modes should be introduced when the definition is clarified. To support this process, we have created a method for introducing modes into a model without modes. Our methods trace data, events, conditions, and functions from system to software requirements, and between different levels of definition.

- (4) Support for reasoning about a target system specification. We have found Allen's theory of temporal intervals to be consistent with SCR concepts, and believe it would be useful for reasoning about temporal order during early specification phases. We have extended predicate logic to include events, so that consistency of mode-transition tables can be analyzed. Engineers using the logic were able to detect inconsistencies in the A-7E Aircraft Software Requirements Specification.

- (5) Extensions to the SCR language. State and event relationships, including decomposition, are defined in the ER model and are used in target system specification. Indexing and predicate qualifiers have been added to the language to simplify specifications involving multiple instances of an object (e.g., aircraft being tracked). Other extensions permit the specification of different timing requirements in different modes, definition of data senescence, and worst case timing for response to stimuli, for environment response, and for computation, communication, and storage access delays.

PFM Formal Model: A PFM model is a definition of a target system that is to be built; it is a statement of the functional requirements. The PFM model augments the SCR model which consists of data templates, function tables, and mode transition tables. These templates and tables are easy to create and understand, but have underlying formalisms. Function tables are extended state machines, where machines operate concurrently with and have state knowledge about other machines.

PFM uses the same templates and tables for model definition. In addition, PFM uses a formal ER model

which supports automation. The ER concept is used to formally define the objects in the PFM model and relationships between these objects. The ER definition can be used as a logical database design for storing target system models and is needed for method automation. The ER model can also be used by the analyst to determine the proper use and power of the PFM method.

In PFM ER diagrams, rectangles represent objects; circles represent two-part relationships; triangles represent three-part relationships; and squares represent four-part relationships. The relationships are only specified in one direction to reduce diagram complexity, but the relationships are complementary. Figure 7 defines a PFM FUNCTION. When relationships are three-part or four-part, a special syntax is used. For example, the three-part relationship (210) reads V ACTIVATED BY VI WHEN I (FUNCTION ACTIVATED BY EVENT WHEN CONDITION). The roman numerals I, V, VI refer to objects which take part in the relationship. The roman numerals appear in the upper right-hand corner of rectangles. We have defined a full set of ER diagrams for PFM.

PFM-Nets: PFM-nets are formally defined and a process transforms a PFM static model of a Target System to a PFM net. If techniques were automated, a PFM-net could be computer generated from the Target System model.

PFM-nets are equivalent to Petri nets with inhibitor arcs, and have the same power to perform system simulation and analysis. Like Petri nets, a PFM-net contains places, transitions, and edges. However, in PFM-nets, each place can represent a condition or an event. Each net contains a set of objects. Each object contains a subset of the places, transitions, and edges in the net. PFM-nets are organized by object, as they are generated from the static model.

The rules for moving tokens in PFM-nets are somewhat different than the rules for Petri nets. An EVENT place keeps its token for one time unit as an EVENT is an instantaneous occurrence. CONDITION places hold their tokens until the associated transition is fired. CONDITION places represent the states of an object. If a CONDITION (state) of object A is linked to a state-transition for object A, the state of object A changes when the transition is fired, and the token is moved from the old state to the new state as in Petri nets. On the other hand, if the state of object A affects a state transition in object B, the state of object A should not change and the token is not removed from the CONDITION place in object A. Figure 8 shows interacting house temperature, motor, and oil valve for a home heating system.

A rapid prototype can be developed from the PFM static model by coding the operational aspects of actions associated with significant objects and events. The rapid prototype consists of the PFM-net generated

from the uncoded portion of the static model, interacting with the coded portion. The PFM rapid prototyping process is effective, according to Bruno and Marchetto's criteria. "Rapid prototyping is an alternative paradigm to the conventional software life cycle... However, the prototyping paradigm is ineffective if it is not supported by a development environment that provides an easy derivation of prototypes from formal specifications and makes the implementation process partially automated" (BRUN 86).

SUMMARY

Existing methods are not adequate for modeling real-time embedded computer system requirements. Eight well known methods were evaluated to determine method strengths and weaknesses, and to determine the optimum basis for synthesizing a more powerful method, a Pragmatic Formal Method (PFM). The best capabilities of existing methods were selected, and where required new capabilities are being created, to ensure that PFM satisfies the needed characteristics for requirements modeling.

FUTURE RESEARCH

Our goal is to develop a method and completion criteria for modeling subsystem boundary properties during the requirements definition phase, so that interoperability can be verified.

Our hypothesis is that currently available modeling techniques can be used to specify subsystem interoperability, but better support is needed:

- Detailed steps have not been defined to support the process.
- Static and dynamic analysis techniques are required to support model integration.
- Methods are needed for specifying and testing feasibility of performance requirements.
- Support is needed for testing the interoperating model.

The philosophy of modeling for interoperability will be on modeling from the boundary of systems, inwards as in the SCR methods and PFM. These methods define system outputs in terms of inputs, conditions, events, and states, providing interdependencies of data, not normally found in an Interface Requirements Specification.

System context will be defined in detail incorporating sequences of data entering and leaving the subsystem, as in the Requirements Driven Design (RDD) method which is based on DCDS (ALFO 91).

Required communication mechanisms for interoperating subsystems will be incorporated into the model as in PAISLEY.

Engineers must specify and allocate requirements for end-to-end timing of critical processing flows, and

specify delays inherent in existing components. Methods developed in our earlier research and by Jahanian and Mok (JAHA 86) will be incorporated into PFM methods and used to document these aspects.

Logic and temporal analysis techniques, developed by Grumman for analyzing consistency when using SCR methods, will be used to ensure consistency of integrated subsystem models.

Dynamic methods are needed to test and integrate subsystem behavior. Researchers have proven that state machines can be mapped to Petri nets, but succinct methods like SCR and STATEMATE use extended state machines. PFM-nets will be further defined and automated to dynamically test behavior and timing of a set of extended state machines.

We will incorporate techniques for checking overriding constraints. Gist (BALZ 82) uses cooperating machines called "demons" to check for conditions that are always prohibited. In a similar manner, Harel (HARE 92) establishes a behavioral specification called a "watchdog" and uses reachability tests to check whether unwanted states can be entered.

Information will be modeled in a maintainable manner, grouping information by objects that occur in the environment and system.

Detailed process steps and completion criteria will be defined.

As a pilot project, boundary properties of subsystems currently under development in our avionics laboratory will be modeled. The boundary properties of several subsystems will be integrated into one consistent model, that will be executed to verify interoperability.

REFERENCES

- (ADCA 85) Ad Cad Ltd., STATEMATE1, The Languages of STATEMATE1, Illogix, Cambridge, MA, 1985.
- (ALFO 91) Alford, M., "Strengthening the Systems Engineering Process," Proceedings of National Council on Systems Engineering (NCOSE), Oct. 1991.
- (ALFO 85a) Alford, M., "SREM at the Age of Eight: The Distributed Computing Design System," IEEE Computer, Vol. 18, No. 4, Apr. 1985, pp. 36-46.
- (BALZ 82) Balzer, R.M., Goldman, N. M., and Wile, D. S., "Operational Specification as the Basis for Rapid Prototyping," ACM SIGSOFT Software Engineering Notes, Dec. 1982, pp. 3-16.
- (BARI 79) Barina, H., Cobey, W., Rosenbaum, J., and White, S., "Automated Structured Design," Proceedings of the IEEE Computer Society's Third International Computer Software and Applications Conference (COMPSAC), Nov. 1979.
- (COAD 91) Coad, P., and Yourdon, E., Object-Oriented Analysis, Yourdon Press, Englewood Cliffs, NJ, 1991.
- (FRAN 87) Frank, G.A., and Thelen, K., "Integrating the Tools of a System Design Environment with a Relational Data Base," Hawaii International Conference on System Sciences, Jan. 1987.
- (HARE 86) Harel, D., "Statecharts: A Visual Approach to Complex Systems," Report CS86-02, Weizmann Institute of Science, Cambridge, MA.
- (HARE 92) Harel, D., "Biting the Silver Bullet: Toward a Brighter Future for System Development," COMPUTER, Vol. 25, No. 1, pp. 8-20.
- (HENI 78) Heninger, K., Kallander, J., Parnas, D.L., and Shore, J., "Software Requirements for the A-7E Aircraft," Naval Research Lab, Washington, DC, Memo Rep. 3876, Nov. 27, 1978.
- (HENI 80) Heninger, K., "Specifying Software Requirements for Complex Systems: New Techniques and their Application," IEEE Trans. on Software Engineering, Vol. SE-6, Jan. 1980, pp. 2-13.
- (JAHA 86) Jahanian, F. and Mok, A.K., "Safety Analysis of Timing Properties in Real-Time Systems," IEEE Trans. on Software Engineering, Vol. SE-12, No. 9, Sept. 1986, pp. 890-904.
- (KMIE 84) Kmiecik, J., and Paul, K., "Tool Specification and Evaluation Alternatives," Grumman ASD Technical Report SRSR-A6-84-003, Aug. 1984.
- (NASH 84) Nash, R., Kaufman, G., and Kmiecik, J., "SCR Methodology User's Manual," Grumman ASD Technical Report SRSR-A6-84-002, Aug. 1984.
- (PNUE 86a) Pnueli, A., "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends," in Current Trends in Concurrency, Overviews and Tutorials, ed. by J.W. deBakker, W.P. deRoever, G. Rozenberg, Springer Verlag, Lecture notes in Computer Science Vol. 224, 1986, pp. 510-584.
- (SHLA 88) Shlaer, S. and Mellor, S.J., Object-Oriented Systems Analysis, Modeling the World in Data, Yourdon Press, Englewood Cliffs, NJ, 1988.
- (TEIC 80) Teichrow, D., Macasovic, P., Hershey III, E. A., and Yamamoto, Y., "Application of Entity-Relationship Approach to Information Processing Systems Modeling," in P. Chen, ed., Entity-

Relationship Approach to Systems Analysis and Design, North Holland, 1980, pp. 15-34.

(TRUX 72) Truxal, J.G., Introduction to System Engineering, Mc Graw Hill, NY, 1972.

(WARD 85) Ward, P.T., and Mellor, S.J., Structured Development for Real-Time Systems, 3 Vols., Yourdon, Inc., NY, 1985.

(WHIT 80) White, S., Meyers, S., and Stumpp, M., "Evaluation of the Software Requirements Tool, PSL/PSA, as Applied to the A-7E OFP," Grumman ASD Technical Report SSD-80-PSLE-00-1, Oct. 1980.

(WHIT 81) White, S., "SRIMP, Software Requirements Integrated Modeling Program," Proceedings of NBS/IEEE/ACM Software Tools Fair at ICSE '81, Mar. 1981.

(WHIT 83) White, S. and Meyers, S., "Software Requirements Methodology and Tool Study for A-6E Technology Transfer," Grumman ASD Technical Report SRSR-A6-83-001, Jul. 1983.

(WHIT 85a) White, S. and Lavi, J., "Embedded Computer System Requirements (REQECS) Workshop," IEEE COMPUTER, Vol. 18, No. 4, Apr. 1985, pp. 67-70.

(WHIT 85b) White, S., "Requirements Modeling for Embedded Computer Systems," Proceedings of Third International Workshop on Specification and Design, IEEE Press, Aug. 1985, pp. 238-241.

(WHIT 86a) White, S., "Two Embedded Computer System Requirements Models: Issues for Investigation," Proc. of International Workshop on the Software Process and Software Environments, ACM SIGSOFT Software Engineering Notes, Vol. 2, Aug. 1986, pp. 106-112.

(WHIT 86b) White, S., "Panel Problem: Software Controller for an Oil, Hot Water Home Heating System," Panel on Software Requirements Methods, Proceedings of COMPSAC, Oct. 1986, pp. 276-277.

(WHIT 87a) White, S., "Method Attribute Checklist," Proceedings of National Conference on Methodologies and Tools for Real-Time Systems, Mar. 1987.

(WHIT 87b) White, S., "A Pragmatic Formal Method for Computer System Definition," PhD Dissertation, Polytechnic University, also published as Tech Report RE-791, Grumman Corporate Research Center, 1992.

(WHIT 92a) White, S., "A Pragmatic Formal Method (PFM) for Computer System Definition and Execution,"

to be published in Proceedings of IEEE International Workshop on Rapid System Prototyping, June 1992.

(WHIT 92b) White, S., et al., "Improving the Practice in Computer-Based Systems Engineering," to be published in the Proceedings of the National Council on Systems Engineering (NCOSE), July 1992.

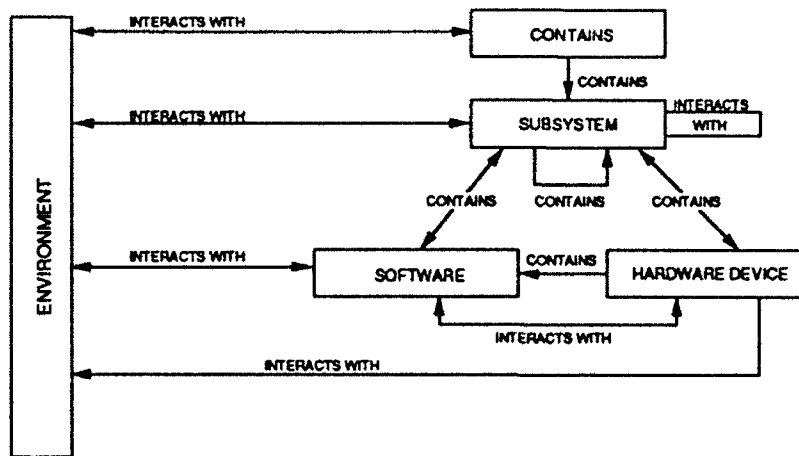


FIGURE 1 DISTRIBUTED EMBEDDED SYSTEM MODEL

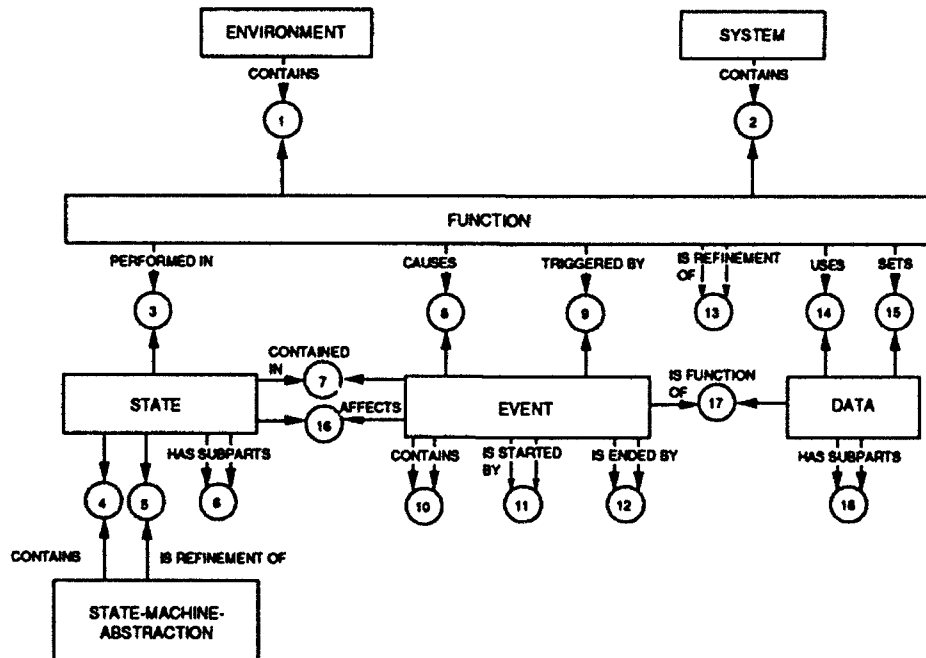


FIGURE 2 GENERIC REAL-TIME MODEL

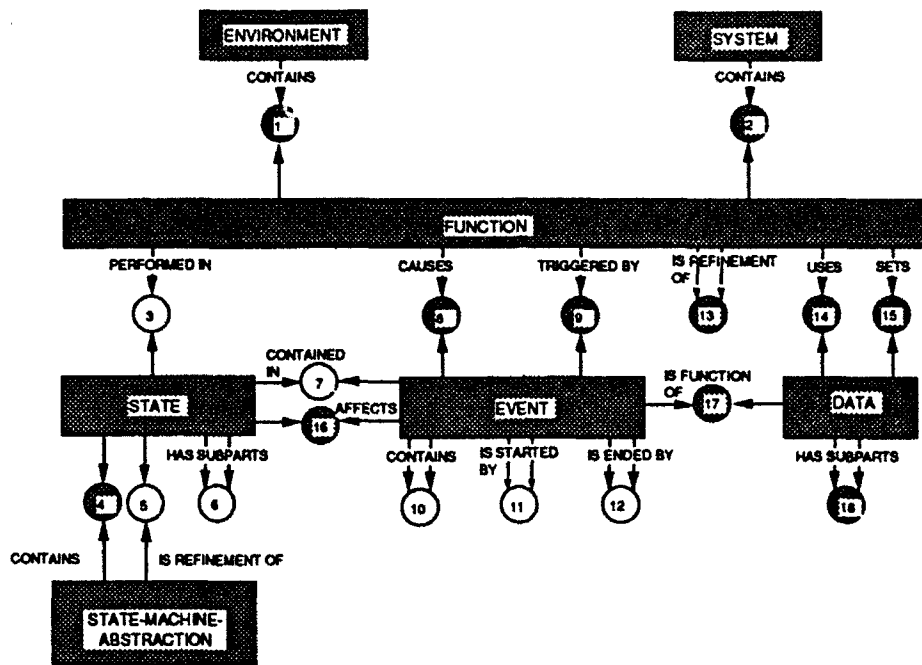


FIGURE 3 DCDS COMPARED TO
GENERIC REAL-TIME MODEL

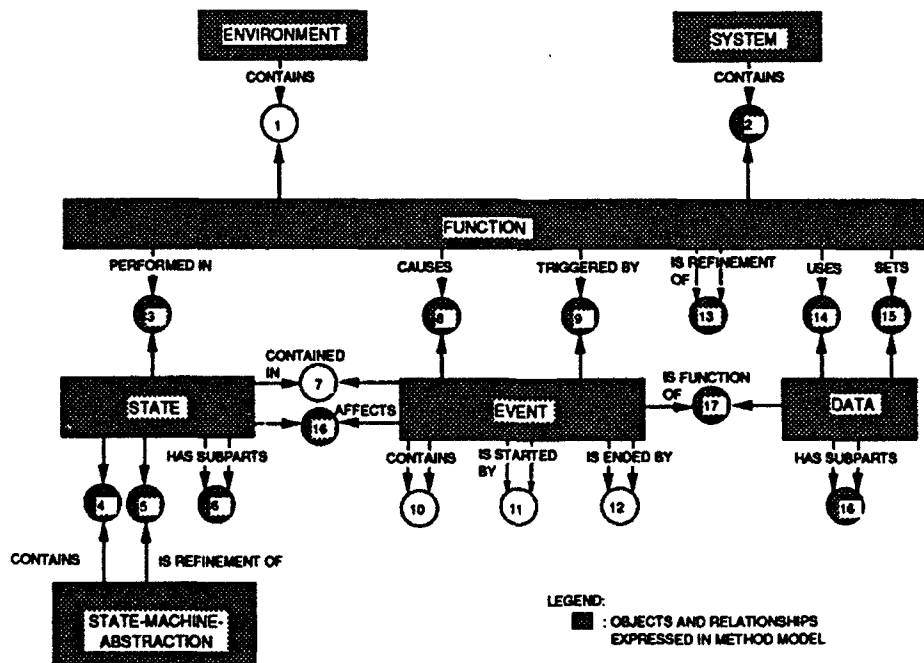


FIGURE 4 SA-RT COMPARED TO
GENERIC REAL-TIME MODEL

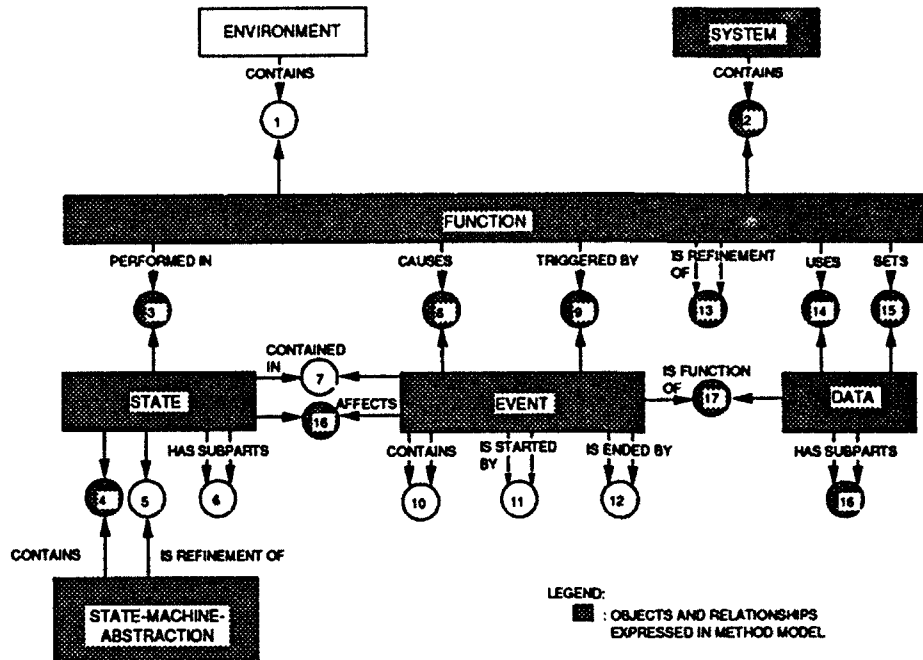


FIGURE 5 SCR/A7E COMPARED TO GENERIC REAL-TIME MODEL

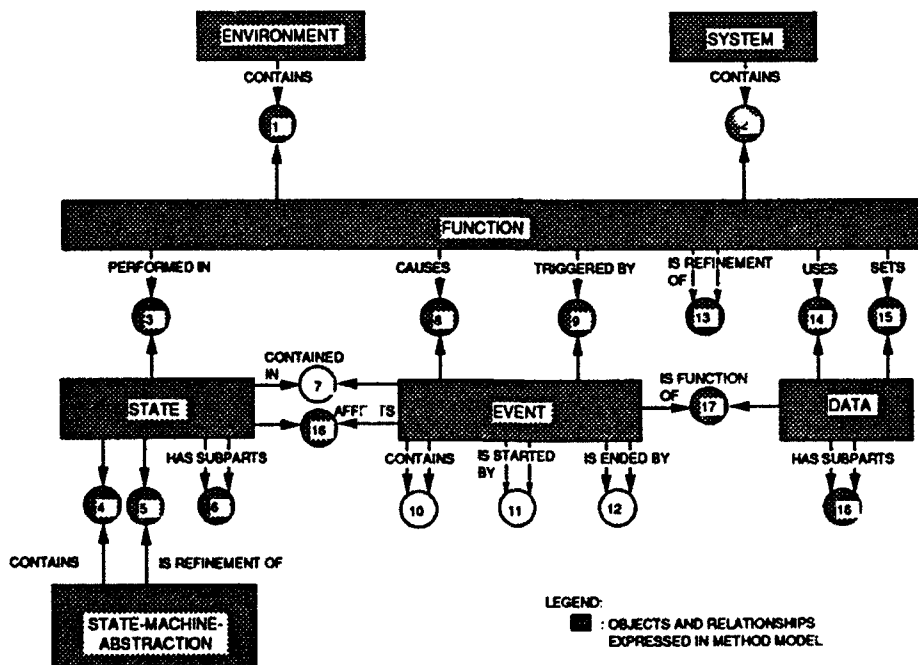


FIGURE 6 STATEMATE COMPARED TO GENERIC REAL-TIME MODEL

	DCDS	HOS	JSD	SART	SCR/AZE	STATEMATE	PAISLEY	PAMELA
PRIMARY OBJECTS (A) FUNCTION (B) DATA, CONDITION (C) EVENT (D) STATE (E) STATE MACHINE ABSTRACTION	(A): NET, FUNCTION, ALPHA (B): INPUT, OUTPUT DATA, CONDITION (C) EVENT (D) SET OF CONDITIONS (E) SYSTEM	(A) FUNCTION (B) INPUT, OUTPUT DATA (C) - (D) - (E) -	(A) PROCESS (B) INPUT, OUTPUT DATA, CONDITION (C) EVENT (D) - (E) -	(A) PROCESS, DFD (B) INPUT, OUTPUT DATA, DATASTORE, CONDITION (C) EVENT (D) STATE (E) STD	(A) FUNCTION (B) INPUT, OUTPUT DATA, CONDITION (C) EVENT, ACTION (D) MODE (E) MODE CLASS, FUNCTION TABLES	(A) ACTIVITY (B) INPUT, OUTPUT, DATASTORE, CONDITION (C) EVENT, ACTION (D) STATE (E) STATECHART	(A) PROCESS, FUNCTION, MAPPING (B) INPUT, OUTPUT, CONDITION (C) ACTION (D) STATE (E) PROCESS CYCLE	(A) PROCESS TASK DFD (B) INPUT, OUTPUT DATA, DATASTORE, CONDITION (C) EVENT (D) STATE (E) STD
FORMAL BASIS (ATTRIBUTE 1)	(1), (2), (3), (5)	(1), (2), (3), (4)	(1), (2), (3)	(1), (2), (3), (5)	(1), (2), (3), (5*), (6)	(1), (2), (3), (5*), (6)	(1), (2), (3), (5)	(1), (2), (3), (4)
CONSTRUCTION PROBLEMS (ATTRIBUTE 2)	PROBLEMS SELECTING PATHS; MODEL DIFFICULT TO CHANGE	PROBLEMS CHOOSING FUNCTIONS; MODEL DIFFICULT TO CHANGE	NETWORK DIFFICULT TO CHANGE	PROBLEMS CHOOSING PROCESSES; MODEL DIFFICULT TO CHANGE	NO PROBLEMS UNLESS MODES ARE USED; MODEL SUPPORTS CHANGE	PROBLEMS CHOOSING ACTIVITIES; MODEL DIFFICULT TO CHANGE	PROBLEMS CHOOSING FUNCTIONS; NOTATION DIFFICULT; MODEL DIFFICULT TO CHANGE	PROBLEMS CHOOSING PROCESSES; MODEL DIFFICULT TO CHANGE
COMPREHENSION PROBLEMS (ATTRIBUTE 3)	PROBLEMS UNDERSTANDING INTER-RELATIONSHIP OF PATHS	PROBLEMS UNDERSTANDING HOW ALL FUNCTIONS WORK TOGETHER ON ONE LEVEL; USE OF DATA ITEMS AS BOTH DATA AND CONDITIONS	PROBLEMS UNDERSTANDING NETWORK FOR LARGE PROBLEMS; HOW PROCESSES WORK TOGETHER; PDL NOTATION	PROBLEMS UNDERSTANDING INTER-RELATIONSHIP OF DATAFLOW DIAGRAMS ON THE SAME LEVEL	PROBLEMS UNDERSTANDING HOW FUNCTIONS WORK TOGETHER; ORDERING OF FUNCTIONALITY	PROBLEMS UNDERSTANDING INTER-RELATIONSHIP OF STATECHARTS AND ACTIVITY DIAGRAMS	PROBLEMS UNDERSTANDING NOTATION	PROBLEMS UNDERSTANDING INTER-RELATIONSHIP OF DFD'S ON SAME LEVEL; CONTROL OF PROCESSES

LEGEND FOR FORMAL BASIS

- (1) INPUT TO OUTPUT MAPPING, (2) FUNCTION COMPOSITION/DECOMPOSITION, (3) PROCESS ABSTRACTION
- (4) DATA ABSTRACTION, (5) COMMUNICATING ABSTRACT MACHINES, (6) COMMUNICATING EXTENDED ABSTRACT MACHINES, (6) PREDICATE LOGIC

TABLE 1 METHOD COMPARISON SUMMARY SHEET 1 OF 3

	OCDs	HOS	JSD	SA-RT	BCRAVE	STATEMATE	PAISLEY	PAMELA
DESIGN INDEPENDENCE (ATTRIBUTE 4)	NOT SUPPORTED	NOT SUPPORTED	NOT SUPPORTED IF POL IS USED	NOT SUPPORTED	SUPPORTED WITHOUT MODES	NOT SUPPORTED	NOT SUPPORTED	NOT SUPPORTED
STEPWISE REFINEMENT (ATTRIBUTE 5)	NET. DATA	FUNCTION. DATA	EVENT	STATE. PROCESS. DATA	TEXT MACRO	STATE. ACTIVITY DATA	STATE. FUNCTION DATA	PROCESS. DATA
SEPARATION OF CONCERNS (ATTRIBUTE 6)	DIFFICULT TO PARTITION MODEL ACCORDING TO CONCERNS OF DIFFERENT READERS	DIFFICULT TO PARTITION MODEL ACCORDING TO CONCERNS OF DIFFERENT READERS	DIFFICULT TO PARTITION MODEL ACCORDING TO CONCERNS OF DIFFERENT READERS	DIFFICULT TO PARTITION MODEL ACCORDING TO CONCERNS OF DIFFERENT READERS	MODEL COULD BE PARTITIONED ACCORDING TO CONCERNS OF DIFFERENT READERS	DIFFICULT TO PARTITION MODEL ACCORDING TO CONCERNS OF DIFFERENT READERS	DIFFICULT TO PARTITION MODEL ACCORDING TO CONCERNS OF DIFFERENT READERS	DIFFICULT TO PARTITION MODEL ACCORDING TO CONCERNS OF DIFFERENT READERS
TIMING CONSTRAINTS (ATTRIBUTE 7)	ASSOCIATED WITH OBJECTS AND VALIDATION POINTS ON PATHS	NOT ADDRESSED	INFORMAL	INFORMAL ASSOCIATED WITH PROCESS	ASSOCIATED WITH FUNCTION	NOT ADDRESSED OTHER THAN TIMEOUT	ASSOCIATED WITH PROCESS PROCESS CYCLE FUNCTION COMMUNICATION BOUNCES OR DISTRIBUTION MAY BE GIVEN	NOT ADDRESSED
OTHER, NON-FUNCTIONAL REQUIREMENTS (ATTRIBUTE 7 CONT'D)	ASSOCIATED WITH OBJECTS	NOT ADDRESSED	NOT ADDRESSED	NOT ADDRESSED	NOT ADDRESSED	NOT ADDRESSED	RELIABILITY	NOT ADDRESSED
SUPPORT'S DESIGN TRADEOFF ANALYSIS (ATTRIBUTE 8)	NOT WELL SUPPORTED	NOT WELL SUPPORTED	NOT WELL SUPPORTED	NOT WELL SUPPORTED	NOT WELL SUPPORTED	NOT WELL SUPPORTED	NOT WELL SUPPORTED	NOT WELL SUPPORTED
IDENTIFICATION OF TESTS (ATTRIBUTE 9)	SUPPORTED BY INPUT TO OUTPUT PATHS	NOT WELL SUPPORTED	NOT WELL SUPPORTED	NOT WELL SUPPORTED	WELL SUPPORTED AT EVENT/ACTION LEVEL	SUPPORTED BY EVENT/ACTION STATEMENTS IN STATE CHARTS	NOT WELL SUPPORTED	NOT WELL SUPPORTED

TABLE 1 METHOD COMPARISON SUMMARY SHEET 2 OF 3

	DCDS	HOS	JSD	SART	SCRUTE	STATEMATE	PAISLEY	PAMELA
CAPABILITY FOR IDENTIFICATION OF TBDS (ATTRIBUTE 10)	FAIR: TBD ASSOCIATED WITH ATTRIBUTES, STUBS	POOR: TBD ASSOCIATED WITH STUBS	POOR: TBD ASSOCIATED WITH STUBS	POOR: TBD ASSOCIATED WITH STUBS	GOOD: TBD ASSOCIATED WITH BLANKS IN TEMPLATES	FAIR: TBD ASSOCIATED WITH BLANKS IN TEMPLATES	POOR: TBD ASSOCIATED WITH STUBS	FAIR: TBDS IDENTIFIED IN ADA PDL
VERIFICATION (ATTRIBUTE 11)	GOOD: STATIC & DYNAMIC TESTS	FAIR	POOR	FAIR	POOR	GOOD: STATIC & DYNAMIC TESTS	GOOD: STATIC & DYNAMIC TESTS	FAIR
REUSEABILITY (ATTRIBUTE 12)	NOT SUPPORTED	SUPPORTED BY ABSTRACT DATA TYPES, NOT BY TREE STRUCTURE	SUPPORTED BY STRUCTURE DIAGRAM, NOT BY NETWORK DIAGRAM OR PDL	NOT SUPPORTED	SUPPORTED BY TABLES FOR DATA, FUNCTIONS	SUPPORTED BY STATECHART, NOT ACTIVITY DIAGRAM	NOT SUPPORTED	SUPPORTED BY ABSTRACT DATA TYPES, NOT GRAPHS
SYSTEM AND SOFTWARE REQUIREMENTS TRACEABILITY (ATTRIBUTE 13)	SUPPORTED	NOT SUPPORTED	NOT SUPPORTED	NOT SUPPORTED	NOT SUPPORTED	NOT SUPPORTED	NOT SUPPORTED	NOT SUPPORTED
FORMAT	GRAPHIC NET, FORMAL LANGUAGE DESCRIPTION	FORMAL DATA TYPE DESCRIPTION, GRAPHICS	GRAPHICS, PDL	GRAPHICS FOR C'D, STD, TEXT FOR MINISPECS AND DICTIONARY	TEMPLATES, DECISION TABLES, TEXT	GRAPHICS, TEMPLATES	FORMAL LANGUAGE DESCRIPTION	GRAPHICS ADA PDL
AUTOMATED TOOL SUPPORT	RSUREVS	HOS TOOLS	JSD SUPPORT TOOLS	CADRE, IDE	NONE	STATEMATE TOOLS	PAISLEY TOOLS	ADA GRAPH

TABLE 1 METHOD COMPARISON SUMMARY SHEET 3 OF 3

ATTRIBUTE SUPPORTED	DCDS	WOS	JSD	SA-RT	SCR/ATE	STATE-MATE	PAISLEY	PAMELA
(1) FORMAL BASIS	2.5		2.5	2.5	5	5	2.5	2.5
(2) MODEL CONSTRUCTION	2.5	2.5	2.5	2.5	5	2.5	2.5	2.5
(3) MODEL COMPREHENSION	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5
(4) DESIGN INDEPENDENCE	0	0	2.5	0	5	0	0	0
(5) STEPWISE REFINEMENT	2.5	2.5	2.5	2.5	2.5	2.5	2.5	2.5
(6) SEPARATION OF CONCERNS	0	0	0	0	5	0	0	0
(7) NON-FUNCTIONAL REQMTS	5	0	0	0	2.5	0	2.5	0
(8) DESIGN TRADEOFF ANALYSIS	0	0	0	0	0	0	0	0
(9) TESTING	2.5	0	0	0	5	2.5	0	0
(10) TBD(S) IDENTIFIED	2.5	0	0	0	5	2.5	0	2.5
(11) VERIFICATION	5	2.5	0	2.5	0	5	5	2.5
(12) REUSABILITY	0	2.5	2.5	0	5	2.5	0	2.5
(13) TRACEABILITY: SYSTEM TO SOFTWARE	5	0	0	0	0	0	0	0
(14) MAXIMAL CONCURRENCY & NONDETERMINISM	0	0	0	0	5	0	0	0
TOTAL	30	15	15	12.5	47.5	25	20	17.5

TABLE 2 METHODS RATED. SUPPORT FOR DESIRABLE CHARACTERISTICS

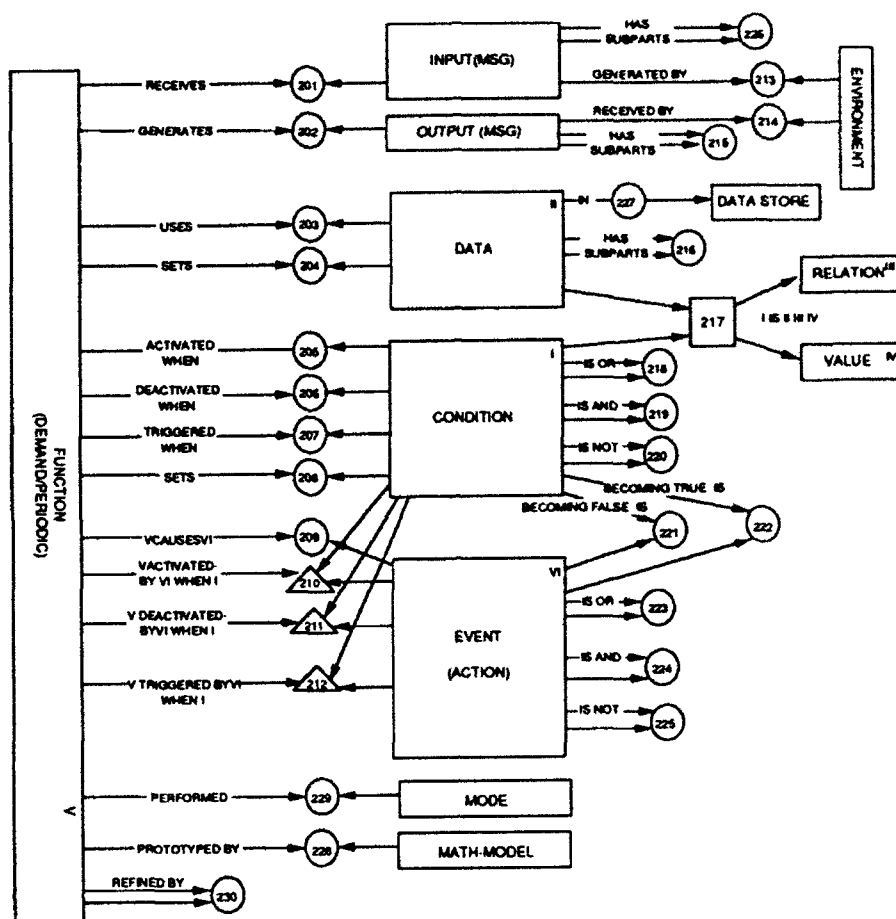


FIGURE 7 PFM FUNCTION RELATIONS, FORMAL DESCRIPTION

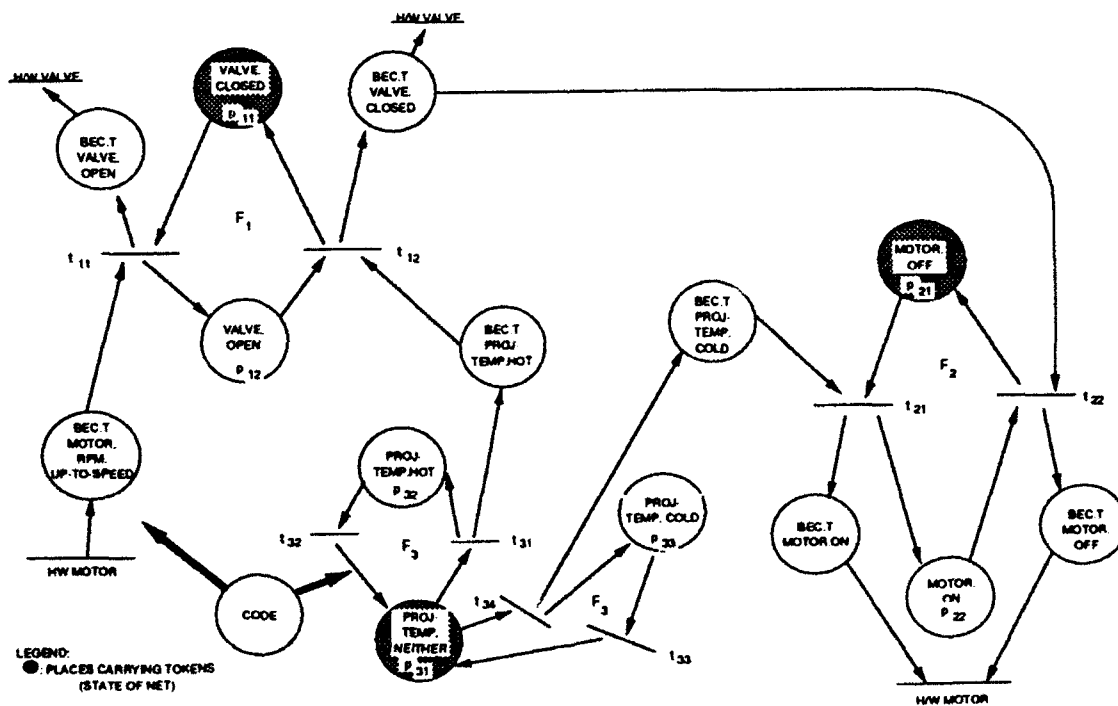


FIGURE 8 PFM-NET GENERATED FROM DEFINED MODEL

IMPROVING THE PRACTICE IN COMPUTER-BASED SYSTEMS ENGINEERING

State of Practice Working Group
IEEE Computer Society Task Force in CBSE

Stephanie White, Chairperson
Grumman Corporate Research Center
M/S A08-35
Bethpage, New York 11714

Mack Alford, Ascent Logic
Brian McCay, MITRE
David Oliver, General Electric
Colin Tully, Colin Tully Associates

Julian Holtzman, University of Kansas
C. Stephen Kuehl, Logicon
David Owens, SPC
Allan Willey, Motorola

Abstract: Problems in the development of large computer-based systems indicate that a new discipline is needed at the systems engineering level. Designing systems with distributed processing and databases requires analysis of critical end-to-end processing flows to determine feasibility and proper allocation. An expanded skills base would be required to enable either software or systems engineers to perform the necessary tradeoff studies concerning software, hardware, and communication components.

This paper informs managers, engineers, educators, and researchers about the need for computer-based systems engineering and the strategic opportunities this discipline provides for systems engineering improvement.

INTRODUCTION

A computer-based system (CBS) includes all system parts that process and control information. Computer-based systems engineering (CBSE) requires that activities of systems engineering be applied to the CBS. CBSE responsibilities include definition of critical processing flows, allocating software to distributed systems, making decisions concerning computer hardware, and sizing communication components.

The IEEE Computer Society Task Force on CBSE was created in 1991 to promote the discipline, encourage research in the field, and establish a framework for education and training (Agrawala, 1991; Lavi, 1991a; Lavi, 1991b). The task force created three working groups: education, research, and practice. This task force paper presents the case for a CBSE discipline, discusses current practices, and identifies market and social imperatives for improving the state of the practice.

The paper addresses the following issues:

- Examples of CBSs and the number of engineers

currently practicing CBSE in the United States,

- Definition of a CBS and CBSE responsibilities,
- Standard and advanced practices, and areas for further research,
- Strategic targets for computer-based systems engineering improvement. These targets can be implemented today, and would have a high benefit-to-cost ratio in terms of process and product improvement. Implementing the suggested improvements should increase corporate profitability and customer satisfaction.

SCOPE OF CBSE

System Types. Systems have become more encompassing, complex, event-driven, physically distributed, and networked. Table 1 shows examples of systems.

Problems with complex and stringent system requirements are reflected in the following examples:

- Space Station Freedom has approximately 1.5 million requirements.
- The Air Traffic Control System requires downtime of no more than six seconds per year for critical functions.
- The next generation of fighter aircraft requires extensive computer control to aid pilot control. Wing and tail control surfaces must be regulated thirty times per second.
- The modern automobile has more computing power than the Lunar Lander when it landed on the moon. It is projected that, by the year 2000, the automobile computer will have more interactive sensor-control loops than the largest chemical refinery in Baytown Texas.
- Computerized systems can cause unsafe conditions, leading to increased requirements to "prove" the safety of system designs.

System Designer Role. A system designer translates requirements into designs, verifies that system behavior meets requirements, allocates functions and behavior to

Table 1. Examples of Systems

Defense	Comm- ercial	Public	Tele- Comm	Auto- motive	Finan- cial
VERY HIGH COMPLEXITY					
Intelligence Fusion	PIMS	Weather Forecasting		Intellig. Highways	
Space Station	Airline Reservations	Air Traffic Control	Central Office Switch		Econometric Model
MEDIUM COMPLEXITY					
Cruiser or B1-Bomber	B-777	BART	Network Control	Dealer Networks	Portfolio Mgmt
Logistics Depot	Mfg Automation	Nuclear Reactor	Campus Backbone	Vehicle Mgmt	NYSE
LOW COMPLEXITY					
Smart Bomb	Cell Control		LAN Control	Cruise Control	Auto-Teller

components, and builds system descriptions. During the development and test phases of a project, a system designer interprets requirements, guides other system designers of related systems, and directs tradeoff studies. Although this effort amounts to only five to ten percent of the total system project, the adequacy, accuracy, and timeliness of this work is critical to the success of all complementary activities.

Number of Computer-Based Systems Designers. The population of system designers must be identified by what they do, rather than by industry, type of system being produced, or even job titles. Ascent Logic Corporation has estimated there were 300,000 people doing system design in the United States in 1988 and it is expected the number will exceed 360,000 by 1996. This is partly from increased employment as forecasted by (Hearings before Joint Economic Committee 1988), but also because more workers are becoming capable of designing systems. Ascent Logic estimates that approximately one-third to one-half of the designers (100K to 150K) are focusing on CBSE. The worldwide population is estimated to approximately double these figures.

ENGINEERING COMPUTER-BASED SYSTEMS

Definition of CBS. A CBS consists of all components necessary to capture, process, transfer, store, display, and manage information. Figure 1 shows a CBS

reference model for a distributed system.¹ This is one of a number of reference models under discussion (Jackson, 1992; Alford, 1991; Oliver, 1991; White, 1991, White, 1987). In the figure, processing entities include analog and digital hardware, firmware, and software. Communications entities provide network services that allow multiple processing entities to exchange information, transparent to application software. Information services provide for exchange of information between processing entities and storage devices, e.g., disks or tapes. Human/computer interaction services including windows, graphics, and command services support interaction between processing entities and people. CBSs interact with the physical environment through sensors and actuators, and also interact with external CBSs.

CBSE. The nature of CBSs requires a different systems engineering knowledge base than that normally required to engineer non-CBSs. All CBSs involve application software and associated services that are conceptual in nature and inherently difficult to grasp. Requirements satisfied by software are frequently ambiguous and subject to change, leading to CBS design changes that may sacrifice system architecture flexibility to ensure performance requirements are met. Furthermore, software changes in complex CBSs can result in unpredictable behavior, both internal and external to the CBS. The distributed nature of a CBS is unique in that CBS resources are frequently geographically dispersed and under the control of different organizations. To exchange data among such systems requires interfaces to describe content, and protocols to describe format.

A dedicated discipline is advocated to address these complex and unique CBS attributes. CBSE as a discipline is analogous to systems engineering in the traditional sense. What differs is the focus, and hence the skills necessary to successfully perform CBSE. CBS engineering is concerned with the following responsibilities:

- Design decisions concerning the distributed nature of the CBS (its architecture),
- Allocation of resources to component developers and management of the coordinated process,
- Allocation of functions and data to CBS resources (processors, software, datastores, displays, Human Computer Interface),
- CBS strategies with respect to safety, security, and fault tolerance,
- Global system management strategy,
- Definition of information services,
- Performance allocations (timing, sizing, availability),
- Testing (component, integration, interoperability with the external environment),

¹This figure is taken from (POSIX 1991).

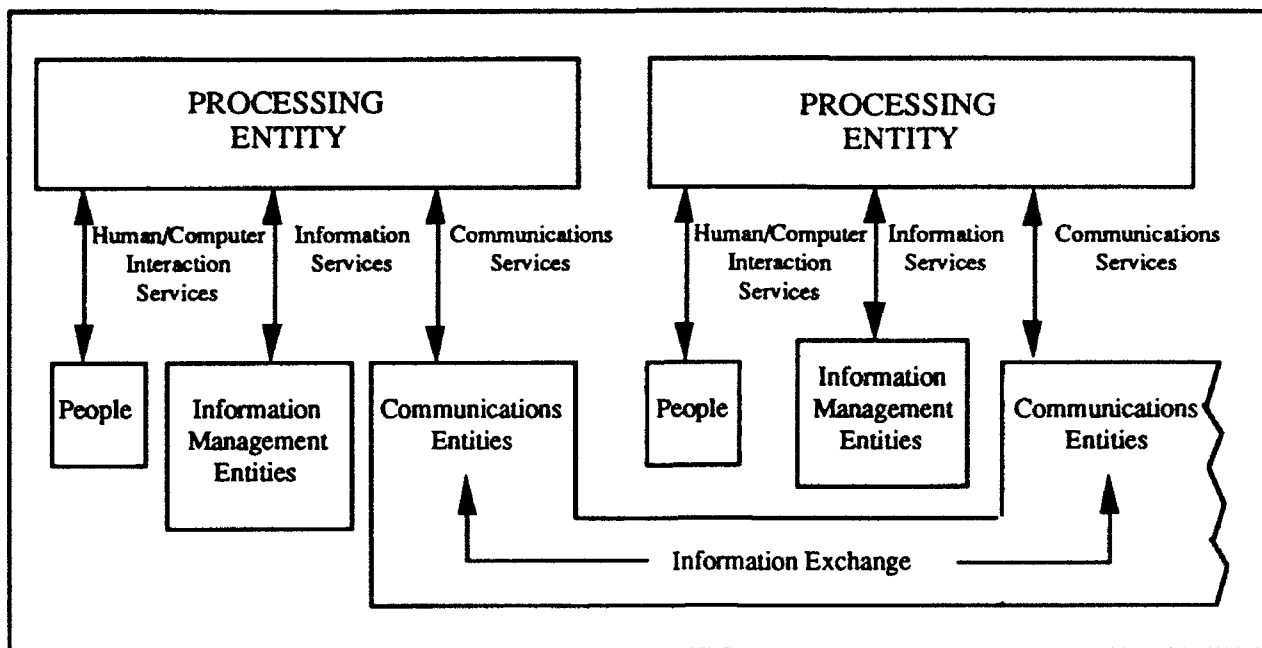


Figure 1. Distributed CBS Reference Model

- Logistics support (maintenance, training, configuration management),
- Implementation of the CBS within the existing environment.

In performing these tasks, engineering tradeoffs must be made, prompted by operational requirements, limited resources (e.g., finances, personnel), CBS component design (e.g., bandwidth, memory size, I/O subsystem, database system), system environment constraints (e.g., operational environment, security measures), and performance thresholds, (e.g., timeliness, throughput, availability).

CBSE STATE OF PRACTICE

This section addresses standard and advanced practice, and topics where further research could result in significant process improvement. Key areas are addressed:

- The CBSE Process,
- Requirements Definition,
- Design (Process and Architecture),
- Interfaces,
- Management,
- Process automation,
- Documentation, and
- Interpersonal Communication.

This description of CBSE state of practice addresses many of the most important issues, but is not meant to cover all issues. The CBSE State of Practice Working Group would like to hear from other CBSE practitioners concerning additional state of practice issues.

CBSE Process. The CBSE process must be tightly integrated with the systems engineering process. Industry has recognized the size and scope of problems with systems engineering processes, as evidenced by the recently formed AIAA Systems Engineering Working Group, the National Council on Systems Engineering (NCOSE), and Europe's Atmosphere Project. Recognition of the need for a special discipline in CBSE is just emerging, as seen by the IEEE Computer Society Task Force on Computer-Based Systems Engineering.

Table 2 summarizes the state of the CBSE process. Numbers in the table refer to paragraphs in the text. The "1" in "1) Undefined CBSE process" indicates that more information can be found in the first paragraph: 1) Process definition.

Table 2. State of CBSE Process

STANDARD PRACTICE	ADVANCED PRACTICE	RESEARCH NEEDED
1)Undefined CBSE process	2)Systems engineering process modeling (may include CBSE tasks)	1)Methods and tools for process modeling 2) CBSE roles and tasks

1) **Process definition.** In general, corporations have not defined their CBSE processes. Some companies use static modeling methods and tools to capture and document their systems engineering

processes while establishing quantifiable process metrics. A few companies use executable modeling tools for process modeling, that for the most part were developed for requirements modeling and were based on state machine or Petri net models. Industry needs better methods and tools for process modeling.

2) **Process modeling needs.** Industry needs a well-defined, executable CBSE process model that incorporates relationships to the overall systems engineering process. The defined CBSE process must be flexible enough to foster process improvement in a timely manner. Industry needs metrics to measure process improvement and product quality.

3) **A CBSE discipline is needed.** CBSE must start at the beginning of the systems engineering process, supporting feasibility analysis and requirements allocation. Allocation decisions are made early in the systems engineering life cycle, sometimes before proposal submittal. These decisions can have significant impact on critical processing flows, requiring analysis of performance and accuracy.

Requirements Definition. Table 3 summarizes the state of practice in requirements definition.

Table 3. State of Requirements Definition

STANDARD PRACTICE	ADVANCED PRACTICE	RESEARCH NEEDED
1) Natural language for systems engineering 1) Inconsistent languages	2) Applying software modeling techniques to systems engineering	1) Consistent language 2) System modeling 3) Model generated scenarios

1) **Language.** Natural language is the standard practice for systems engineering specification. Most software engineers use software requirements models, frequently modeling data, data flow, and control flow. Hardware engineers develop models using VHDL(s). The use of inconsistent languages by different disciplines leads to communication and traceability problems. Information passed from systems engineering to other specialties must be complete and must be in the target methodology and notation. Information must be transferred without manual reentry. Attention needs to be paid to both the semantic issues and the tool interface issues involved in this transition. Changes must be propagated in both directions.

2) **Methods and models.** More advanced practitioners are applying software engineering methods to systems engineering. These methods are not sufficient to support all CBSE functions. Practitioners

need effective methods for specifying system performance requirements that support system design and derivation of software performance requirements. They also need useful paradigms that promote reuse of existing requirements specifications, and use of non-developmental items (NDI). Analysis for completeness, consistency, and correctness is primitive. Tools should apply logic, numerical analysis, and domain understanding to the analysis problem.

3) **Operational scenarios.** Practitioners need models for generating a wide range of operational scenarios, including many with low probability. They need these scenarios to determine whether the requirements are consistent and adequate in terms of defining and constraining the behavior of the system within its environment.

Design Process. Table 4 summarizes the state of the CBSE design process.

Table 4. State of CBSE Design Process

STANDARD PRACTICE	ADVANCED PRACTICE	RESEARCH NEEDED
2) Static functional models 2) Hard coded dynamic models	3) Planning for change 4) Starting decision capture	1) Systems/software dialog 2) Support for tradeoffs 4) Decision analysis 5) Reuse

1) **System/software engineering dialog.** In too many cases, systems engineers do not understand what information should be provided to CBS implementers. Information is provided in the wrong sequence, and is not analyzed to sufficient detail. The dialog with the implementer is frequently not structured, e.g., providing the correct level of information to determine feasibility versus the correct level to design.

2) **Static and dynamic models.** Engineers, using standard software practices, produce static functional models that provide various representations for human review and analysis. These engineers seldom use dynamic modeling techniques such as Petri nets. They frequently hard-code dynamic models of fixed-point designs, and use few parameters to support requirements changes and tradeoff analysis. They have a limited ability to trade capability (data flow and logic control) versus resource utilization and performance. They have done little research on nonlinearities caused by scale-up of capability/data, and seldom analyze or model scale effects.

3) **Planning for change.** System developers are beginning to use open system architectures, as engineers plan for change. CBSs can change

significantly during their life cycle, and researchers addressing an improved process must address this fact. New processes must handle multiple variants for a system ("system families") efficiently. Developers must trace the implication of design changes to/from higher level specifications and across families. To support change analysis, CBSE needs effective design decision capture.

4) **Ramifications of systems engineering decisions.** Systems engineers make "high-level" or "architectural" or "system-wide" design decisions. These are policy decisions that should inform and constrain subsequent design and management decisions relating to various subsystems. It is unclear how to present and propagate these key decisions, or how to monitor subsystem design decisions to ensure that they are not in conflict with system-level design decisions. In addition, many high-level or system-level decisions result in major consequences to the CBS. Engineers do not understand these consequences when the decisions are made. Research is needed to determine what types of decisions have major consequences, what are the ramifications of these decisions, and how such decisions should be made.

5) **Specification for Reuse.** Industry needs effective models of system classes and of common subsystems/components that are used across classes. These models are important for supporting domain analysis and effectively storing high-level system segment or subsystem descriptions in a library. To effectively use a library, it is important to be able to assert: "what I want is exactly like that except for...."

Design Architecture. Table 5 summarizes the state of CBSE architecture.

Table 5. State of CBSE Architecture

STANDARD PRACTICE	ADVANCED PRACTICE	RESEARCH NEEDED
1)Performance at expense of architecture 2)Component hierarchies, few guidelines	2)Standard architectures, open systems	2)Domain architectures 3)Empirical data relating methods to quality designs

1) **Performance at expense of architecture.** In standard practice, design emphasis is on system performance at the expense of other architecture issues. These performance optimized solutions are inflexible and hard to adapt to changing requirements.

2) **Component hierarchies, standard architectures.** Designers usually decompose unprecedented systems into component hierarchies using few guidelines. Standard domain architectures are appearing as building blocks for precedent systems.

3) **Empirical data unavailable.** Designers do not sufficiently use partitioning rules associated with maintainable systems, so there is not enough analytical data to validate these rules.

Interfaces. Table 6 summarizes the state of interface definition.

Table 6. State of CBSE Interface Definition

STANDARD PRACTICE	ADVANCED PRACTICE	RESEARCH NEEDED
1)Documented by data element description and protocol	2)Environment and subsystem documentation and simulation	1)Interface modeling 2)Capabilities for modeling host systems 3)HCI partitioning

1) **Level of abstraction needed.** Engineers normally document interfaces by data element description and protocol, which is inadequate. Research is needed to define the level of abstraction that supports modeling the boundary properties of a set of subsystems, to verify that their combined behavior matches the system requirements. If this can't be done, a system must be built and tested before it is known whether it meets the requirements.

2) **Modeling host systems.** Engineers need capabilities for modeling host systems, to predict the effects of the designed system on the host. Host systems are frequently human activities systems, within which our designed systems are to be used.

3) **Human/computer partitioning.** When engineers design systems, they implicitly specify the tasks of system users. Designers need better knowledge of how to partition functions and responsibilities between people and designed systems. Designers should not assign functionality to a system just because it is technically feasible, or even cost effective. They need better human-computer interface (HCI) models. They usually do not model this interface adequately since there is no defined process for integrating the knowledge of all stakeholders.

Management Table 7 summarizes the state of CBSE management.

Table 7. State of CBSE Management

STANDARD PRACTICE	ADVANCED PRACTICE	RESEARCH NEEDED
1) Good cost data for precedented systems	2) Domain specific WBSs 3) Risk management 4) Data collection	4) Process and product characterization for data collection

1) **Costing.** Different cost models result in different cost estimates, using gross overall metrics based primarily on lines of code. These cost models apportion total cost and effort estimates to each life-cycle phase; engineers need fine grain metrics for each phase. Industry can usually estimate the cost of precedented jobs but still has problems costing unprecedented jobs.

2) **Work Breakdown Structure (WBS).** Large projects normally track cost and schedule against a WBS. If the WBS and CBS architecture are inconsistent, the WBS cannot support CBS status tracking. This inconsistency frequently exists, since engineers define the WBS before they know the computer system architecture. To solve this problem, industry needs WBS(s) that support each CBS application domain.

3) **Risk management.** Successful projects perform risk management, but use primitive methods. Generally engineers do not analyze detailed data from previous programs to understand cause and effect. Some risk assessment tools exist, but they do not interact with requirements and design tools, where engineers identify risk issues. Industry needs general tool suites that support risk management views.

4) **Data collection.** In good practice, engineers collect data to manage the current project. In best practice, they use data to improve engineering processes and make predictions for future projects. Research has provided some capabilities for collecting software sizing data and using the data to size new software components. CBSE needs better techniques for process and product characterization, and industry must collect and have access to the data it knows how to characterize.

Process Automation. Table 8 summarizes the state of CBSE process automation

1) **Levels of tool support.** There are several layers of CBSE tool support. At the lowest functional level, automated tools accomplish a specific task in a particular phase of the system life cycle. An automated requirements capture tool is an example of such a tool. At the next level, tools assist the systems designer accomplish many tasks across many phases of the life cycle. A requirements tracing tool provides this level of

support. Further, there are tools and techniques designed to support management and integral processes across all system life-cycle phases. Examples of these tools are problem tracking and reporting tools and project tracking and reporting tools. At the very highest level, there is a class of tool that provides a "framework" into which individual tools are deployed appropriately over the life cycle, and that maintains all records and documentation for system development and operation. Analogous concepts are found in CAD with the "CAD framework initiative" and in software with "Integrated Project Support Environments."

Table 8. State of CBSE Process Automation

STANDARD PRACTICE	ADVANCED PRACTICE	RESEARCH NEEDED
1) Task oriented, some tool interfaces, software environment	1) Frameworks, approach to repetitive tasks	1) Integrated Systems/CBSE environment 2) Efficient change mgmt

2) **Higher level tools needed.** Unfortunately, there are few examples of sophisticated, high-level process automation tools that systems engineers widely use. A recognition of the need exists, but tool suites have not kept pace with this recognition. An area where such tools are needed is in change management. Efficient change entry is a major problem for large complex systems. Major changes may require updating information about a single entity in several places in databases of a number of tools. This multiple manual update process is both costly and error prone. A standard model/schema underlying all tools in use is a desirable solution.

Documentation. Table 9 summarizes the state of CBSE documentation.

Table 9. State of CBSE Documentation

STANDARD PRACTICE	ADVANCED PRACTICE	RESEARCH NEEDED
1) Natural language specs	1) Databases and documentation generators	1) Integral role with process 2) Designed approach

1) **Integral role with process.** Industry tends to focus attention on those documents that are delivered to the customer, but each and every artifact of the process is an element of system documentation. Engineering drawings, test reports, software source code, requirements tracings, defect tracking reports, and many

other such "documents" are a part of what we do. To date, industry has focused little attention on the integral role of capturing all these artifacts as part of the CBSE process. Significant change will occur only when industry integrates methods and tools to support improved processes throughout the CBSE life cycle.

2) **Designed and automated approach.** In part, the weak role that process automation has played in CBSE is to blame. Without necessary databases and automated tools, control and maintenance of documentation is tedious, repetitive, and prone to human error. The introduction of general purpose tools such as word processors and electronic spreadsheets has contributed to improved performance in this area. A "through-designed" approach is needed into which these general purpose tools would be fitted.

Interpersonal Communication. Table 10 summarizes the state of interpersonal communication.

Table 10. State of Interpersonal Communication

STANDARD PRACTICE	ADVANCED PRACTICE	RESEARCH NEEDED
1)Diverse team: Same syntax/ different semantics	1)Concurrent engineering 4)Training	2)Defined roles 3)CBSE as integral part of systems engineering

1) **Diverse team.** Successful application of the systems engineering process hinges on the abilities of a diverse team of specialists to communicate with a common viewpoint. This is especially true when the product is a CBS, as diversity of backgrounds adds complexity to the communication process itself.

2) **Undefined process.** Software engineers, as detail design engineers, perceive that their interface with systems engineers is poorly defined. The problem lies in a lack of precise definition of the allocation process and the failure to trace software specifications to the top-level functionality of the system.

3) **Component view.** Another cause of difficulty is the view that software, computer hardware, and communications assets are component pieces of the systems engineering discipline rather than a whole. Systems engineers find it difficult to break out of this partitioning paradigm, and software engineers are not apprised of the tradeoffs and design decisions.

4) **Cross-training needed.** Academic and industry in-house training programs for systems and software engineering must take a more interdisciplinary view, share some common courses, and work toward the development of a common set of semantics.

ACHIEVABLE SYSTEMS ENGINEERING IMPROVEMENTS

Developing and using complex systems requires major capital investment. Industry must make investments in large systems wisely, with thorough consideration of what the system is to do, the rewards for doing it, and an investigation of feasibility. The systems engineer must ensure that he/she has thoroughly captured user needs and defined a feasible system design that meets all system requirements and constraints.

Industry must act expeditiously to improve systems engineering by advancing CBSE practice. Corporations can implement each of the following suggestions today.

CBS Modeling. Various modeling approaches augment text specification with semantically precise representations for engineering information. There is a gap between current text-based practice and future model-based practice that must be closed quickly and economically. Closing the gap means an extensive culture change and substantial retraining.

Retraining efforts must target methods and notations that engineers can readily learn, as retraining costs usually dominate transition costs when implementing new methods. If at all possible, new notations and methods should evolve gracefully from notations, methods, and concepts currently in use.

A Defined Process. The engineering process includes a sequence of process steps, and policies for process control, documentation, and staffing. Industry should define the process in layers to separate engineering steps from alternative methods of control, documentation standards, and staffing choices. Layers of description may include:

1. Description of process steps.
2. Description of the control process and organizational groups or boards that perform control.
3. Description of representations used for information captured at each step of the process.
4. A mapping of each step and representation onto tools that automate that step.
5. Description of the staffing of engineering alternatives.
6. Description of the review process and information used for review.

Industry should base the systems engineering process on a set of models that define engineering information in a way that computers can capture for interpretation, execution, consistency, and correctness. The process should support one-time entry of information, both for new designs and for changes.

Dynamic Analysis and Simulation. Industry should capture the behavior of systems in a representation that can be executed dynamically for analysis. This ability

gives rigor to the system description. Developers should apply the same representation to scenarios that describe how the system interacts with its environment. In addition, industry should use simulation to prove feasibility and develop benchmarks.

Performance Requirements Management. Traditionally, engineers budget performance requirements to components as they decompose the system. They budget quantities such as weight, power, heat production, heat dissipation, reliability, time response, and memory size from a parent component to subcomponents. They should use methods and tools to track these budgets, and should compare design, simulation, and test results to budgeted specifications.

Metrics, Costing, and Tracking. Metrics, costing, and tracking are essential both for short-term decisions and for long-term continuous process improvement. Well-defined and broadly accepted metrics are required to quantify the work. Industry needs cost models to transform these metrics into numbers for particular applications. Corporations must track work, both to control activities and to calibrate cost models.

Scalability. Closing the gap between current practice and mature practice involves serious issues of proof of scalability. Engineers must try new methods and tools on modest-sized systems, so that method deficiencies can be eliminated and unanticipated benefits can be incorporated.

Major Risk Factors. Investment choices are difficult in these times of strong competition. Organizations must make decisions at a time when there is no solid baseline of methods, metrics, and cost for the systems engineering process. They can estimate anticipated cost improvement, but cannot derive it from existing data.

CONCLUSIONS

Industry needs a CBSE discipline to integrate component views during specification and design of critical end-to-end processing flows. Such a discipline would encompass knowledge of the systems engineering process, software, digital and analog processing hardware, and communications. The combined discipline is necessary for making proper CBS design tradeoffs, decisions, and allocations. Many engineers are performing CBSE tasks, but there are no defined positions, tasks, and roles. The discipline is necessary for process improvement, and for fostering research and training.

AUTHORS

Authors practice, research, and teach in the CBSE discipline:

Dr. Stephanie White, Principal Engineer, Software Process, Grumman Corporate Research Center; Mack Alford, Chief Scientist, responsible for the systems engineering tool RDD, Ascent Logic Corporation; Dr. Brian McCay, Lead Engineer for large C³ Systems, MITRE; Dr. David Oliver, Systems Engineer, GE Corporate Research and Development; Colin Tully, consultant with Colin Tully Associates; Dr. Julian Holtzman, Professor of Electrical and Computer Engineering, and Director of the Center for Excellence in Computer Aided Systems Engineering, University of Kansas; C. Stephen Kuehl, responsible for systems engineering process improvement for V-22 Post Deployment; Logicon Strategic and Information Systems; David Owens, Senior Systems Engineer, Software Productivity Consortium; Allan Willey, in charge of process improvement, Cellular Infrastructure Group at Motorola.

REFERENCES

- Agrawala, A. K., Lavi, J. Z., and White, S. M., "Task Force on Computer-Based Systems Engineering holds first meeting," *IEEE Computer*, Vol. 24, No. 8, August 1991, pp. 86-87.
- Alford, M., "Strengthening the Systems Engineering Process," *Proceedings of NCOSE*, October 1991.
- Hearings before the Joint Economic Committee, U.S. Congress, "Employment in the Year 2000: A Candid Look at Our Future," Supt. of Docs., Congressional Sales Office, U.S. G.P.O., Washington, D.C., 1988.
- Jackson, K. et al., "Standards for Conceptual and Design Models of Computer-Based Systems," Eindhoven Workshop, February 1992, to be published.
- Lavi, J. Z., "Formal establishment of Computer-Based Systems Engineering urged," *IEEE Computer*, Vol. 24, No. 3, March 1991.
- Lavi, J. Z. et al., "Computer-Based Systems Engineering Workshop," *Proceedings of SEI 1991 Conference on Software Engineering Education, Lecture Notes in Computer Science*, Number 536, Springer, 1991.
- Oliver, D. W., "Model Based Systems Engineering," *Proceedings of CBSE Workshop*, March 1991, to be published.
- Posix Open System Environment, IEEE Standard, Draft 14, November 1991.
- White, S. M., "Foundations for Computer-Based System Definition," *Proceedings of CBSE Workshop*, March 1991, to be published.
- White, S. M., *A Pragmatic Method for Computer System Definition*, Ph.D Dissertation, Computer Science, Polytechnic University, Brooklyn, NY, June 1987; also Tech. Report RE-791, Grumman Corporate Research Center, 1992.

INTEGRATION II

EXCHANGE OF INFORMATION BETWEEN DESIGN CAPTURE AND DESIGN OPTIMIZATION TECHNIQUES: THE DESTINATION INTERFACE SPECIFICATION

by

Insup Lee, Evan Lock, Rajesh Purushothaman, and Moon Lee

Abstract

This paper describes the DESTINATION Interface Specification (DIS). Design Structuring and Allocation Optimization (DESTINATION) is an ongoing research project at the Naval Surface Warfare Center (NSWC) to provide a new methodology for design optimization and trade off analysis of real time systems. The need for DIS arises from the inherent adaptiveness of the DESTINATION system to a wide range of source and target tools. DIS not only allows DESTINATION to coexist with various systems but also dictates standards for a comprehensive way of capturing design information. The basic structure of DIS reflects a method of extracting/incorporating design information that is otherwise not available across a collection of tools. DIS accommodates the identification of additional design information, allowing for customization of the source and target tools. The focus of the DIS research and development work is currently in the area of system logical modelling and implementation modelling.

1. Introduction

One of the primary thrusts behind the Systems Design Synthesis project of the NSWC's Engineering of Complex Systems Technology Block Research Program (ECS) is to provide a new methodology for systems engineers in the area of Design Optimization and Trade-Off Analysis. Systems engineers require such a new methodology to cost effectively construct and maintain increasingly complex mission-critical, real-time systems.

Application complexity has increased not only due to functional demands, but also because of technological advances. The present and future combat systems must respond to an expanding theater of commands, as well as the requirement to perform in an integrated manner. Technologically, the advent of parallel computers and high speed networks opens many opportunities to provide greater defense capabilities. These functional and technical factors greatly increase the design space that the systems engineer must explore in search of a design that satisfies all requirements. The idea behind design optimization and trade-off analysis is to provide the systems engineer with the necessary tools and techniques to systematically evaluate and exploit the vast design space.

DESTINATION is the name given to the NSWC research effort that focuses on developing the necessary tools and techniques to support such a methodology for design optimization and trade-off analysis [HoNH]. The emphasis on this project is design structuring and resource allocation tools and techniques. The design structuring involves making decisions regarding decomposition/recomposition and fragmentation/defragmentation of hierarchical designs. Resource allocation includes the mapping of logical design objects onto implementation resources in a near-optimal manner.

The need for an interface specification to perform design optimization first arose on a predecessor project to DESTINATION called EDA or Expert Design Advisor [HoHN]. The first version of the interface specification, developed for EDA, was used to standardize the format of

inputs for the development of four resource allocation optimization algorithms - Gantt [Pear], Data-Oriented, Genetic [Davi91] [Gold], and Simulated Annealing [KiGV]. Reuse of the same data structures reduces the size of the development effort and increases the ability to adapt to new algorithms. The need for the exchange of information between various front-end case tools has initiated the development work on standards for the interface specification that would enhance portability, adaptability, maintainability and extensibility for a wide range of source/target tools.

To better understand the function and structure of the DESTINATION Interface Specification, it is necessary to further explain the associated DESTINATION methodology. By reviewing the context diagram in Figure 1, we can see the DESTINATION methodology's scope and contribution within the systems life cycle.

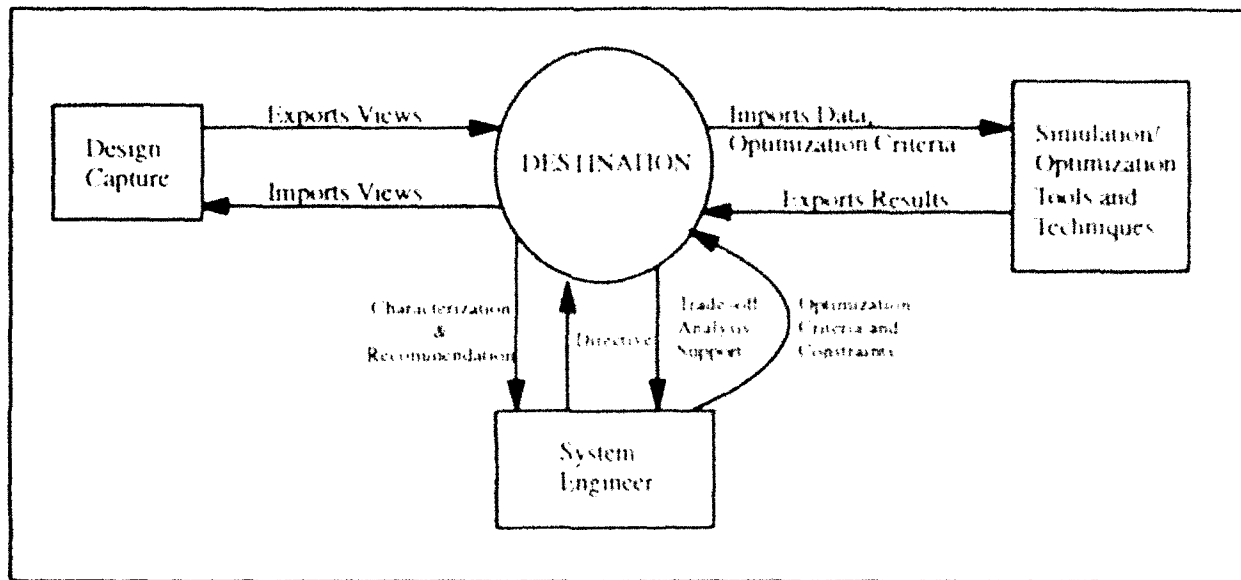


Figure 1: DESTINATION Context Diagram.

The human systems engineer plays a critical role within the methodology. The systems engineer's input is fundamental for selecting the subject of design optimization and evaluation, applying analysis techniques and interpreting results. The methodology supports the systems engineer by making characterizations and recommendations. The systems engineer may accept or override these outputs.

A complete scenario of steps can be mapped into the context diagram to describe the methodology.

1. The systems engineer gives a directive to select a design capture view for analysis.
2. DESTINATION interacts with the system's engineer to determine the following:
 - a. System characterization.
 - b. Formulation of design goals.
 - c. Application of design rules.
 - d. Recommendation for use of simulation/optimization tools and techniques.
3. The system's engineer directs DESTINATION to import the necessary data for the selected simulation/optimization tools and techniques.

4. Results from simulation/optimization are exported to DESTINATION for evaluation.
5. When satisfied with the achievement of design goals, any modification to the design capture views are imported to the Design Capture system to maintain consistency. Although these steps are listed sequentially, it is expected that there will be a high degree of iteration within and among these steps, particularly when performing trade-off analysis.

Many approaches have been followed that specify what to capture, such as, structured analysis and design [WaMe], object-oriented design [BOOC], and how to capture it. One of the most robust approaches to be defined, which is consistent with earlier DESTINATION research efforts, was jointly developed by NAVSWC and Trident Systems Corporation ([Karc], [Hoan]). This approach to forward design capture represents one set of information that may be incorporated into the DESTINATION methodology for analysis. Though DESTINATION is not restricted to any particular Design Capture approach, the NAVSWC/Trident approach is one of the most robust and places the strongest demands on DESTINATION, so it is advantageous to use from a research perspective. Furthermore, use of this approach insures integrated results within the System Design Synthesis project of the ECS block program.

Basically, the forward design capture accepts design information according to three models: the conceptual model, the logical model, and the implementation model. Each is described below.

The conceptual model captures the operational ideas of the system from the perspective of the operational environment and information modelling. The environmental view establishes the conditions and environment in which the system must operate including a description of the system architecture's scope and boundaries, test plan, and operational scenarios. The conceptual model allows the system engineering team and the customer to form a clear understanding of the subject system.

The logical model includes a description of the functional and behavioral views of the system, without regard for any particular implementation decisions. The emphasis within this design capture model is on what the system should do as opposed to how it should do it. The behavioral view provides an understanding of the system from a dynamic perspective.

The implementation model documents the hardware, software and human resources which represent a particular embodiment of the system under design. The hardware architecture describes the physical resources of the system including the components, interconnection topology and protocol, and rationale for selection. The software architecture describes the Computer Software Configuration Items (CSCIs) and the executable software tasks including the messages passed between tasks. The human resource description includes the number of personnel required to operate the system under various conditions and the level of training and experience for each operator.

There is no restriction on what design methodologies may be used within the development of any of the three models.

Likewise, any number of simulation/optimization tools and techniques are available for use within DESTINATION. Optimization algorithms that may be applicable for use include computation/communication-oriented, genetic search and simulated annealing. Simulation

techniques that may be interwoven with the optimization algorithms include petri-net simulation (e.g. SES/workbench, ADAS), queueing theory [CCCC], and general purpose simulation languages (e.g. Simscript). Future advances in both optimization and simulation are to be expected.

2. Requirements of the DESTINATION Interface Specification (DIS)

As described in the previous section, the DESTINATION Interface Specification is the layer of data structures and export/import routines that permit application information to flow in and out of DESTINATION. From one perspective, DIS bridges the design capture facilities with optimization decisions and from another perspective, DIS integrates the execution of simulation models and optimization algorithms with design evaluation and recommendations. This iterative path of capturing, modelling, evaluating and recommending becomes significantly more streamlined by having a consistent and robust medium of exchange.

A number of requirements impacted the development of DIS. These requirements can also be viewed as motivating factors for making the investment in DIS.

1. Tool Independence

It is desirable to have a methodology be independent of a particular toolset. There may be financial and training constraints that oppose acquiring a toolset, particularly when a comparable one may already be in place. In the context of design capture, for instance, there are several Front-End Computer-Aided Software Engineering (FE-CASE) tools that are operational within Department of Defense (DOD) programs, most notably, Cadre's Teamwork and IDE's Software Through Pictures (StP). The interface to DESTINATION should handle data from Teamwork just as easily as data from StP.

Certainly, as part of the access routines into the FE-CASE system's repository, there will be some effort that is not reusable. The goal is to minimize this effort. Figure 2 shows how this is done in the context of interfacing with Cadre's Teamwork.

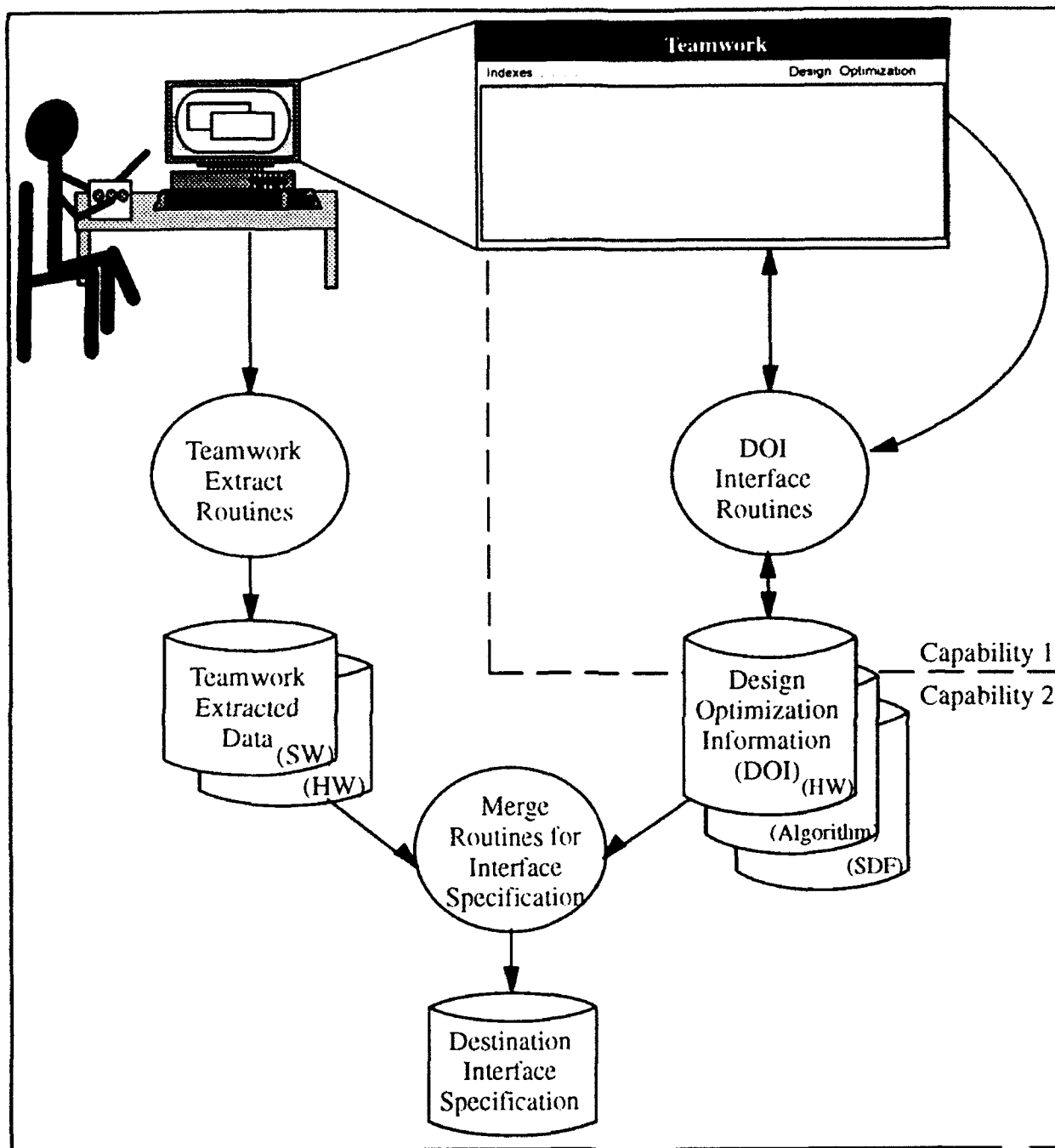


Figure 2: Design Capture Interface.

Two capabilities are shown in Figure 2. The first capability provides the systems engineer with the capability to supplement the design capture process with information for design optimization. There are three types of design optimization information (DOI) that have been included as part of the design capture supplement: System Design Factors [HoNH], data required by optimization algorithms and hardware resource descriptions and characteristics. The second capability extracts information stored directly in the Front-End CASE system's repository representing the information contained within the CASE graphics (bubbles, flows,

connections, etc.) The two information sources, the Front-End CASE dependent data and the Front-End CASE independent DOI, are then merged to create a DIS compatible file.

A similar, though possibly more complicated, choice of tools to develop interfaces exists within the Simulation System/Optimization Technique domain as in the FE-CASE area.

2. Implementation Independence

The DESTINATION toolset contains many interconnected subsystems, such as for design characterization, design evaluation, and for making recommendations regarding design structuring and resource allocation. Development of these subsystems can proceed more independently by sharing the DIS among them.

Furthermore, developers of algorithms for resource allocation, scheduling, and design structuring can utilize the DIS as a departure point for their innovation. DESTINATION then provides a convenient proving ground for determining the situations where the algorithm performs best. This makes for a win-win situation for DESTINATION and algorithm developers: there will be an increased likelihood that their algorithms will be transferred to practical use and likewise DESTINATION's library of algorithms on which it bases its optimization recommendations will progressively expand. It is expected that the algorithm developers will, in turn, uncover additional requirements for the DIS and through feedback DIS will progressively improve.

3. Supports Incomplete Information

If optimization is to be performed on a bottom-up basis, there may be substantial information that may not have been provided on a higher level. To proceed under these circumstances, default values can be associated with the DIS data structures and be supplied as required by the optimization algorithms.

4. Gain Wide Acceptance

DIS must be designed so that it can be used widely, as described in Figure 1, by systems engineers, algorithm developers, tool vendors and standards bodies.

5. Transportability

DIS should facilitate the transportability of design capture, design optimization and simulation information from one computer environment to another.

6. Uniformity and Cohesiveness

The DIS model should be simple and uniform, while minimizing the amount of concepts, types and classes of operations.

7. Implementability

Vendors of simulation systems, front-end CASE systems, and algorithm developers should be able to utilize DIS with only a reasonable effort. The design of DIS should allow for flexibility in implementation while maintaining consistent operational semantics.

8. Extensibility

As mentioned above, tool vendors, algorithm developers and systems engineers will uncover additional requirements on DIS. DIS should not preclude any extensions to its scope to satisfy evolving needs.

9. Performance

The DIS design must allow for efficient operation from both external access of design capture and simulation systems as well as from internal DESTINATION procedures.

To satisfy these requirements, several basic design decisions for DIS were made.

- i. Represent the data structures for easy mapping onto a flat ASCII file. This accommodates the requirements for acceptability, transportability, implementability, extensibility, and performance.
2. Utilize Ada as the language for formally specifying DIS. There were several underlying reasons for this:
 - a. Ada is a DOD standard.
 - b. Ada is widely available on many computing platforms.
 - c. Ada's package facilities and specification/body separation could be used to express multiple layers of abstraction.
 - d. Ada was very successful in its use as a specification language for the Ada Semantic Interface Specification (ASIS) definition [BISp]. ASIS is a vendor-independent, non-proprietary bridge between Ada libraries and Ada tools.

There are a number of alternative methods for specifying the interface. English was dismissed as being too ambiguous. Use of formalisms like the Backus Naur Form (BNF) and Extended Backus Naur Form (EBNF) has the advantage of concise accuracy allowing little room for ambiguities and vagueness but does not allow high level representation. C was not used because of its lack of abstraction facilities. C++ and Common Lisp Object System (CLOS) are viable alternatives, particularly due to their strong object orientation and inheritance facilities. Presently, they lack the standardization and DOD acceptance of Ada. There are systems and associated languages that specialize in interface definition and actually automatically generate some code necessary for declaring and accessing the interface [NEST]. These facilities warrant further investigation, but are not DOD standards.

There are a number of potentially useful integration standards that are emerging, such as PCIS, Case Integration Services (CIS), IRDS, CASE Document Interface Format (CDIF), IEEE-P1175 and NGCR's PSSWG [StSh]. None of these efforts, however, are directly working in the area of design optimization and lack representation of much needed information. Through planned participation with these working groups and standards bodies DIS should beneficially impact these efforts.

3. Components of the Interface Specification

3.1. Overview

The current version of DIS, 2.0, is divided into several packages at its top level. Each of these packages is comprised of lower level packages. This basic structure, at present, is shown in Figure 3.

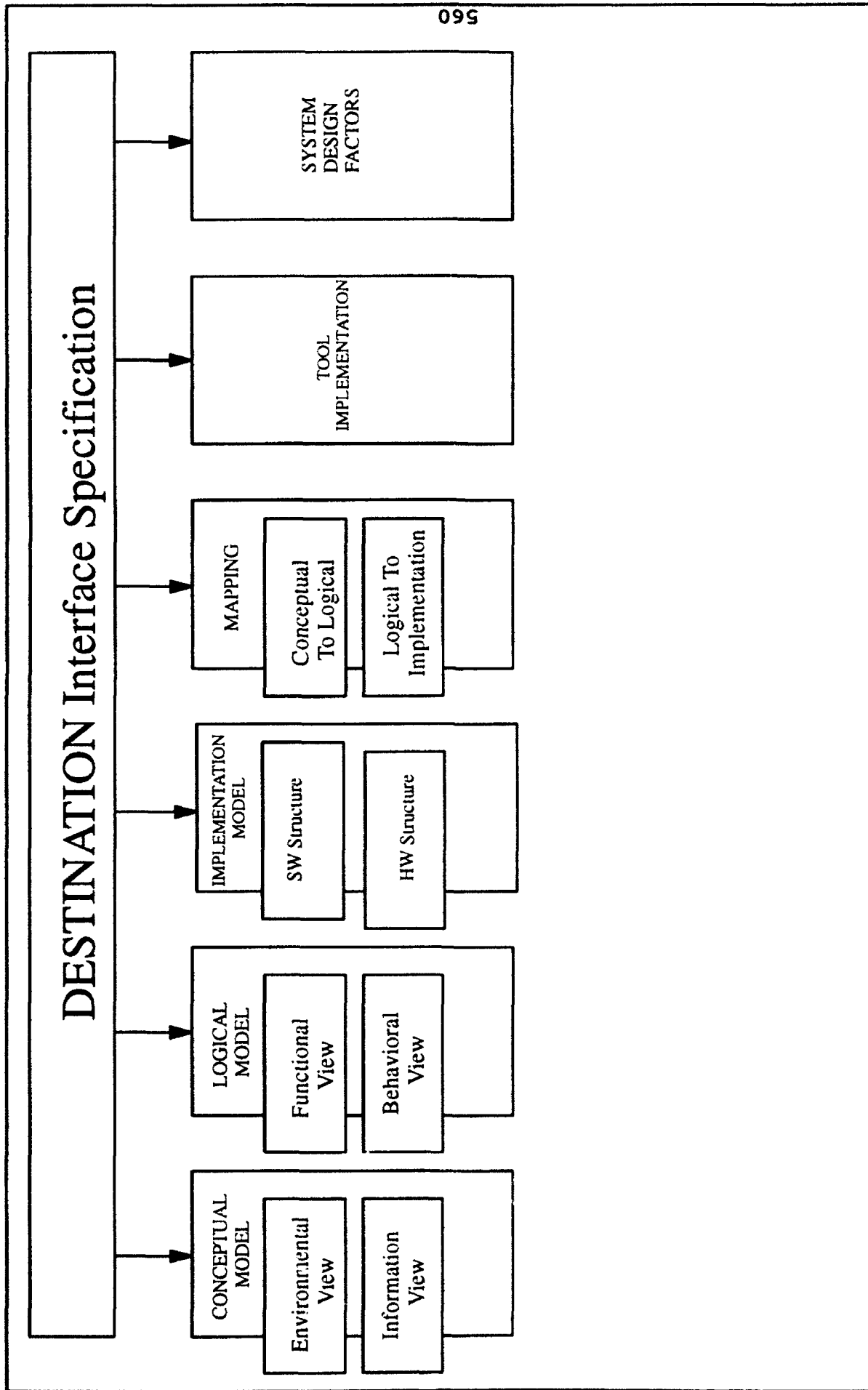


Figure 3: DESTINATION Interface Specification.

The following section describes the implementation model in greater detail with brief descriptions of the structural components and the Ada data type declarations for some of the major components. This is where the emphasis of the current effort has been. Development of a technical report describing the other packages is in progress.

3.2. Implementation Model

The implementation model contains data representations for making and analyzing decisions for resource allocation and design structuring. Representations are needed for the following information:

1. A task graph to depict the candidate software configurations.
2. A resource graph to depict the candidate hardware configurations.
3. Constraints derived from requirements. Constraints are presently divided into two categories: placement constraints and timing constraints. These constraints impact the effectiveness of optimization. Each one of these types of constraints contain several sub-types of constraints that will be described further below.

The implementation model of a real-time system consists of one or more implementation views (i.e., a list of implementation views). The Ada type declaration for this information in the DIS is shown in Figure 4. Each implementation view consists of a software structure diagram, a hardware structure diagram and a list of mappings of software components to hardware resources.

```
type DIS_implementation_view_type is
  record
    DIS_sw_structure_diagram      : DIS_sw_structure_diagram_ptr;
    DIS_hw_structure_diagram      : DIS_hw_structure_diagram_ptr;
    DIS_implementation_mapping_list : DIS_mapping_view_ptr;
    DIS_implementation_view_next   : DIS_implementation_view_ptr;
    DIS_implementation_view_previous : DIS_implementation_view_ptr;
  end record;
```

Figure 4: Implementation View Components. DIS_Implementation_view_previews

The term *structure diagram* is used to reference a collection of directed graphs, drawn with respect to a selected methodology, that captures information about a set of components and their relations along with any hierarchical decomposition. For example, a tree of data flow diagrams may be considered as one type of *structure diagram*.

The list of mappings is provided, since for the same software and hardware structure diagrams, we may apply different allocation tools to determine possible mappings. Each of the components of the implementation view are further described below.

To better explain the data structures represented within the implementation model, graphic figures have been provided. Figure 5 contains a legend of the graphical notations. In Figures 6, 10, and 11, there are three types of edges connecting the data structure:

1. A *points to* relation to indicate that a structure contains a variable which points to another data structure. There are two types of *points to* relations (edges). The first type represents decomposition through a *contains* or *is contained by* relations depending on the direction of the arrow. Typically, the *contains* relation points downward (vertically) on the page and the *contained by* points upward (vertically) on the page. The decomposition implies that when a data structure contains another data structure, the former data structure may be viewed as the parent and the latter is called a child. The second type denotes a *has* relation and does not involve a notion of decomposition into child entities.
2. A *references* relation when two structures show a common data element.
3. An *is linked to* relation showing that the data structure is part of a linked list. For each of these relations there are two pointers—one for the next occurrence in the list and one for the previous occurrence in the list. This double linked list structure allows for reduced programming in traversing the list.

3.2.1. Software Structure

Figure 6 represents the software structure architecture. The data types for the software structure is shown in Figure 7. Each software structure diagram is represented by a list of modules and a list of edges between modules. A *module* represents a collection of nodes and edges within a *structure diagram* (as explained above). Again, using the data flow diagram as an example, a *module* could be considered as a level of decomposition.

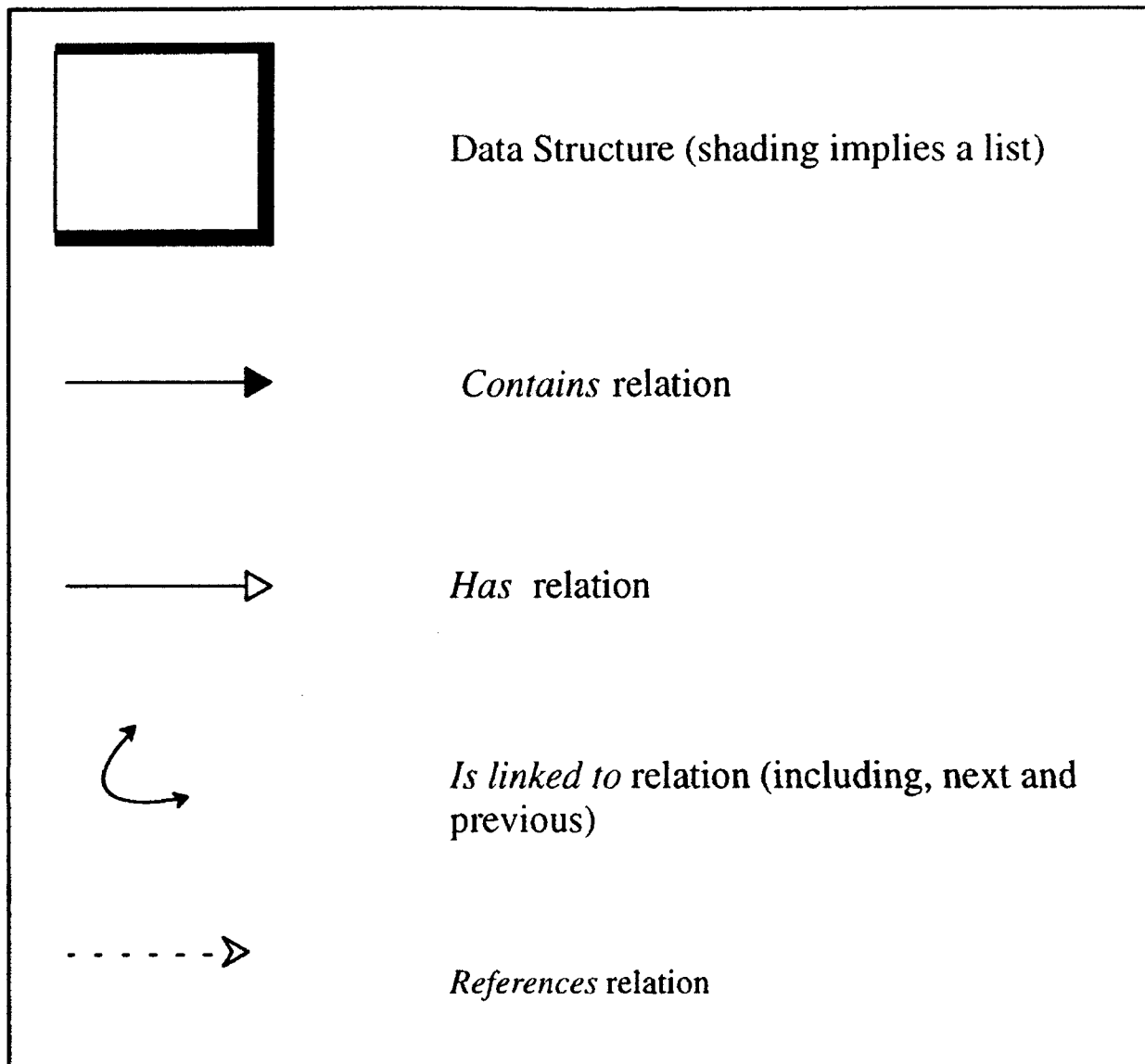


Figure 5: Nodes and Edges Used to Describe DIS Components.

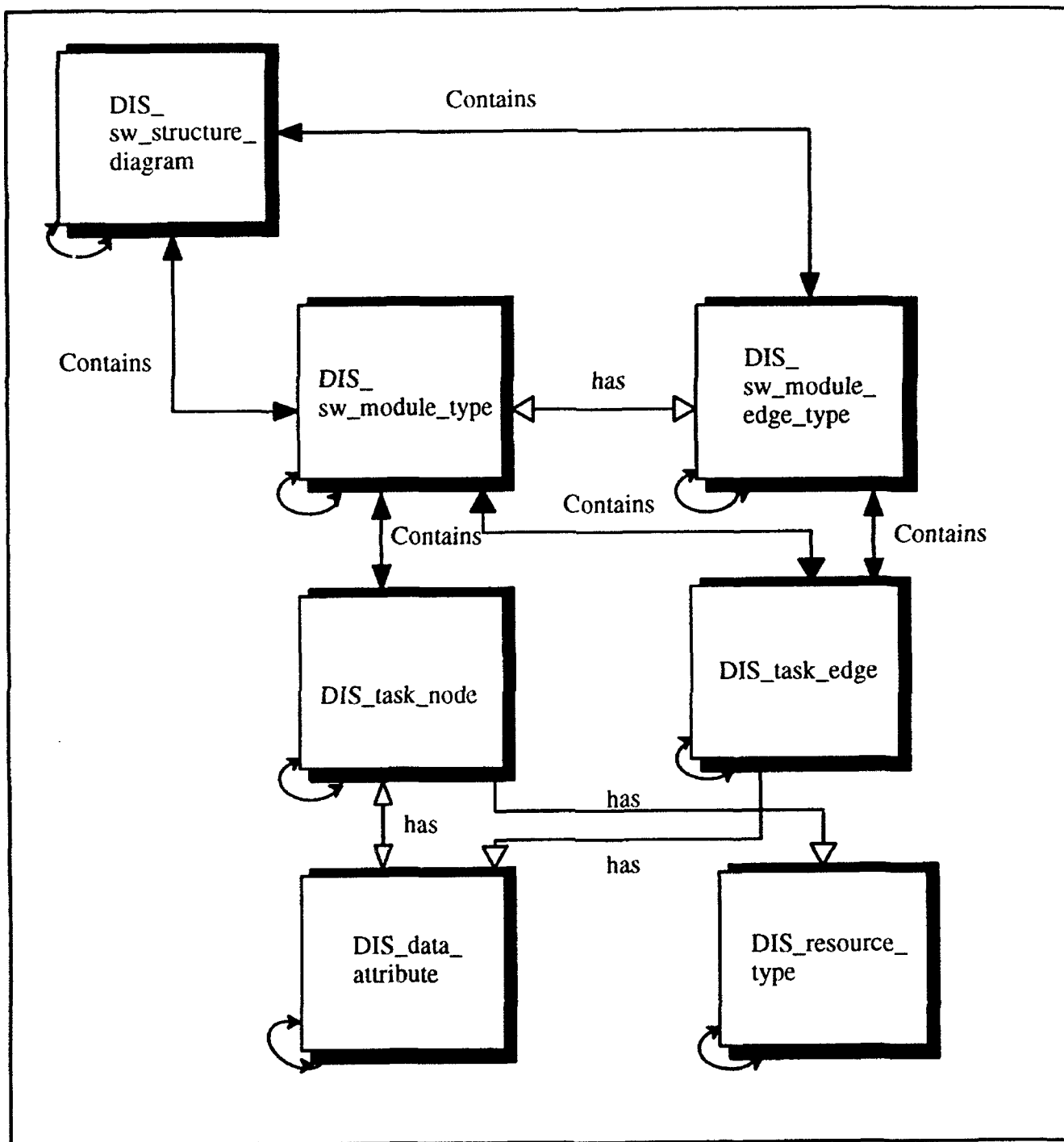


Figure 6: Software Structure Architecture

```

type DIS_sw_structure_diagram is
  record
    sw_structure_diagram_name      : DIS_sw_diagram_name_type;
    sw_structure_diagram_id        : DIS_sw_diagram_id_type;
    sw_module_list                  : DIS_sw_module_ptr;
    sw_module_edge_list            : DIS_sw_module_edge_ptr;
    sw_structure_diagram_next      : DIS_sw_structure_diagram_ptr;
    sw_diagram_previous            : DIS_sw_structure_diagram_ptr;
    parent_implementation_view     : DIS_implementation_view_ptr;
  end record;

```

Figure 7: Software Structure Data Type.

Software modules can be nested and each module includes its own task graph. Task graphs cannot be nested since the node of a task graph cannot be a module. However, nested relations between tasks can be captured using nested modules. Our view is that the task represents a separately executable computational entity.

A software module contains the following information:

1. The hierarchical, sibling and nesting relations between modules.
2. The identity of task graphs that belong to the module. In addition, there are two special kinds of edges (called `entry_super_edge` and `exit_super_edge`). They are used to identify the entry and exit points of the task graph at the module level.

The data type for a software module is shown in Figure 8.

```

type DIS_sw_module_type is
  record
    module_id                      : DIS_sw_module_id_type;
    module_name                    : DIS_name_type;
    parent_sw_structure            : DIS_sw_structure_diagram_ptr;
    parent_module                  : DIS_sw_module_ptr;
    next_module                    : DIS_sw_module_ptr;
    previous_module                : DIS_sw_module_ptr;
    submodule_list                 : DIS_sw_module_list_ptr;
    -- define links between super edges of the submodules.
    sw_module_edge_list            : DIS_sw_module_edge_ptr;
    -- Task graph belonging to this module.
    task_node_list                 : DIS_task_node_ptr;
    task_edge_list                 : DIS_task_edge_ptr;
    entry_super_edge_list          : DIS_task_edge_ptr;
    exit_super_edge_list           : DIS_task_edge_ptr;
  end record;

```

Figure 8: Software Module Data type.

A task graph is a directed graph: each node denotes a schedulable computational entity and an edge represents a precedence relation between two nodes. For each task node in the

DIS_task_node structure, there is a *task_input_list* to identify input data and *task_output_list* to identify output data generated by the task. In addition, *task_predecessor_list* identifies tasks that execute before the task and *task_successor_list* identifies tasks that execute after the task. There is an *and_or* flag associated with the above four task lists that specifies whether all input (or output) data are needed (or generated) by the task. This information is required by some optimization algorithms. Each task may include timing information such as ready time, deadline and duration. In addition, it identifies resources it needs. For resource needs, *DIS_resource_type* identifies the resource a task needs and the amount it needs. For each task edge, *task_edge_data* identifies the data associated with the edge along with the duration of availability of the data. In addition, *from_task_node* and *to_task_node* specifies the source and destination of the edge.

The declarations for the task node data structure and the task edge data structure are shown in Figure 9.

```

type DIS_task_node is
  record
    task_id                : DIS_task_id_type;
    task_name               : DIS_name_type;
    task_structure          : DIS_task_structure_type;
    task_description        : DIS_task_description_type;
    -- Task data dependencies.
    task_input_and_or       : DIS_and_or_type;
    task_input_list         : DIS_data_type_ptr;
    task_output_and_or      : DIS_and_or_type;
    task_output_list        : DIS_data_type_ptr;
    -- Task precedence relations.
    task_predecessor_and_or : DIS_and_or_type;
    task_predecessor_list   : DIS_task_node_ptr;
    task_successor_and_or   : DIS_and_or_type;
    task_successor_list     : DIS_task_node_ptr;
    -- Timing information.
    task_ready_time         : DIS_time_type;
    task_deadline           : DIS_time_type;
    task_duration           : DIS_time_type;
    -- Resource needs.
    task_resource_needs     : DIS_resource_ptr;
    task_max_replication    : DIS_task_count_type;
    task_buddy_task         : DIS_task_node_ptr;
    task_priority           : DIS_task_priority_type;
    task_execution_probability : DIS_task_e_probability_type;
    task_communication_delay_matrix : DIS_task_comm_matrix_ptr;
    task_error_cumulation   : DIS_task_error_type;
    task_imprecise_error_convergence : DIS_task_error_type;
    task_next               : DIS_task_node_ptr;
    task_previous           : DIS_task_node_ptr;
  end record;

type DIS_task_edge is
  record
    task_edge_id           : DIS_task_edge_id_type;
    task_edge_data          : DIS_data_type_ptr;
    from_task_node          : DIS_task_node_ptr;
    to_task_node            : DIS_task_node_ptr;
    next_task_edge          : DIS_task_edge_ptr;
    previous_task_edge      : DIS_task_edge_ptr;
  end record;

```

Figure 9: Task Node and Edge Structure.

3.2.2. Hardware Structure

A hardware structure diagram defines a hardware configuration. A hardware configuration is viewed as consisting of hardware nodes, connected by hardware links. Each node is recursively viewed as consisting of internal nodes that are connected by internal links. The architecture of the hardware structure diagram is shown in Figure 10.

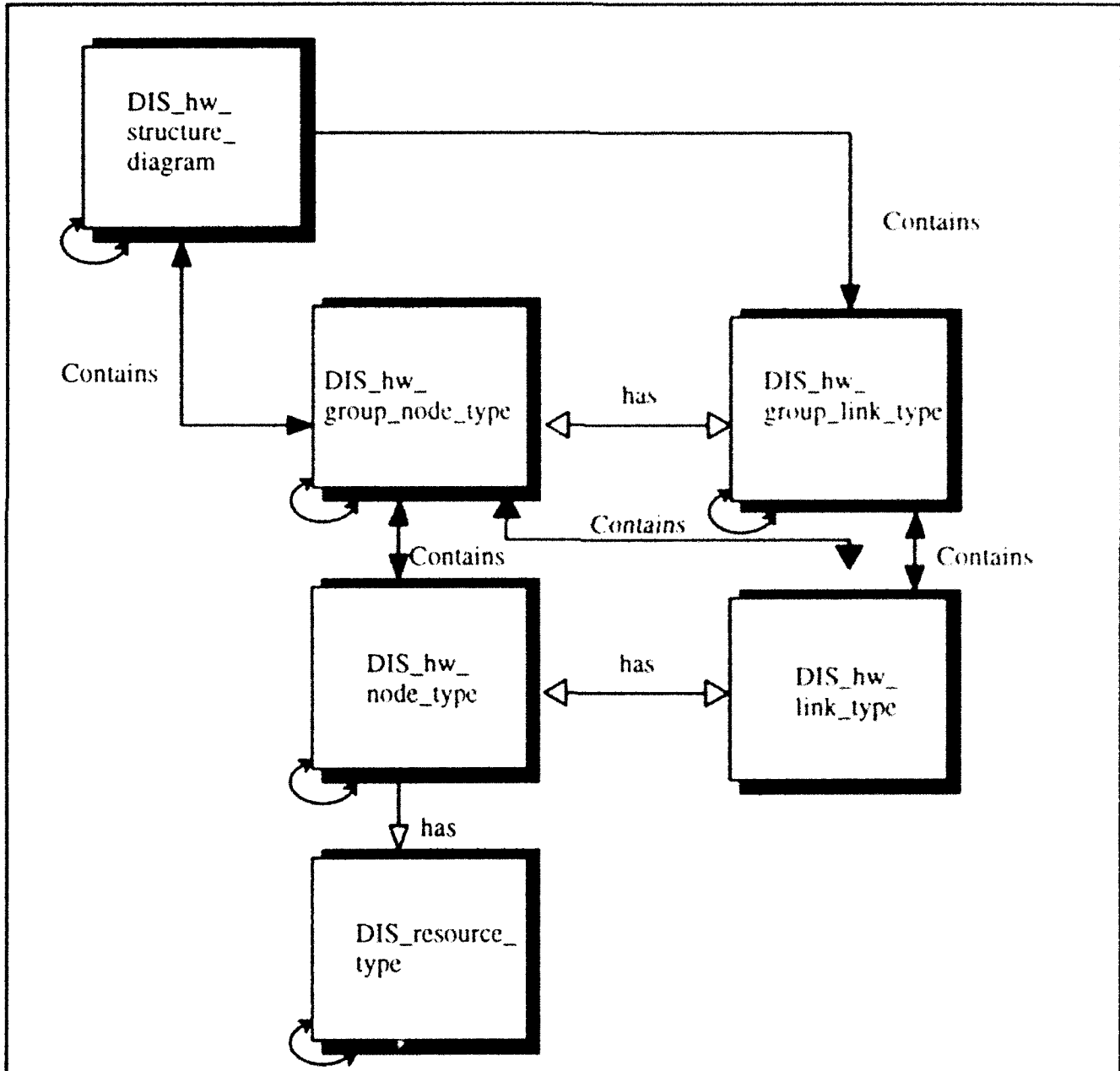


Figure 10: Hardware Structure Architecture

3.2.3. Mapping Structure

A mapping assignment consists of mapping constraints and task assignments. Figure 11 represents the mapping structure architecture. There are two types of mapping constraints: timing constraints and placement constraints. Each mapping constraint includes a preference value that specifies the importance of meeting the mapping constraint; the magnitude of the value reflects its importance.

The data structure for mapping constraint is shown in Figure 12. There are four kinds of timing constraints, each timing constraint is defined on a set of tasks:

1. *complete_within t* means that the set of tasks should complete within t time units of each other.
2. *start_within t* means that the set of tasks should start within t time units of each other.
3. *complete_path_within t* means that the sequence of the set of tasks should complete within t time units from the beginning of the sequence.
4. *complete_start_within t* for two tasks, A and B, means that B should start within t after the completion of A.

There are three kinds of placement constraints, each placement constraint is defined on a set of tasks:

1. *place_together* means that the set of tasks should be assigned to the same hardware component.
2. *place_separate* means that the set of tasks should be assigned to different hardware component.
3. *place_at* means that the set of tasks should be assigned at a particular hardware component.

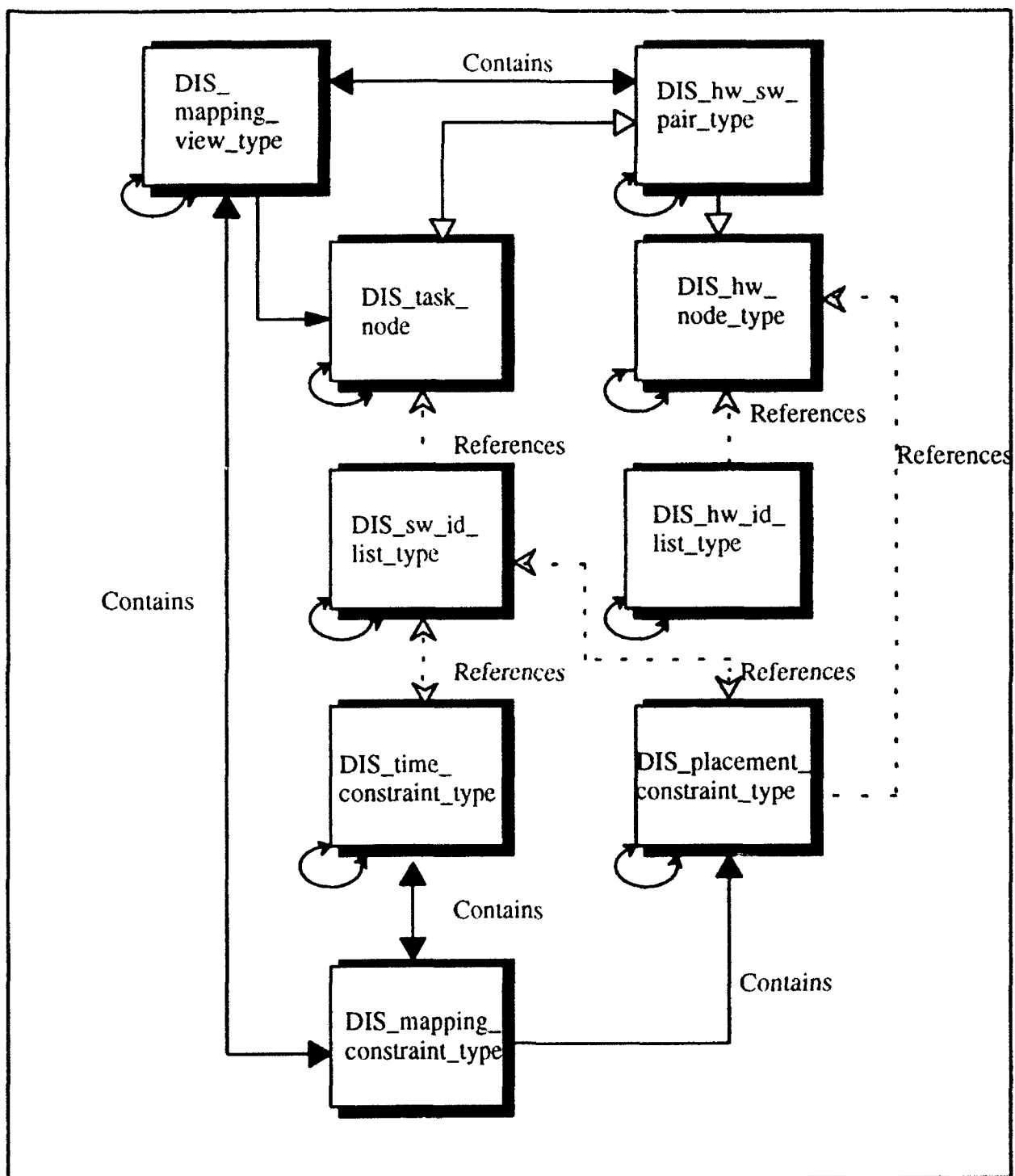


Figure 11: Mapping Structure Architecture.

```

type DIS_mapping_constraint is
  record
    timing_constraint          : DIS_t_constraint;
    placement_constraint       : DIS_p_constraint;
    parent_mapping_view       : DIS_mapping_view;
  end record;

type DIS_t_constraint_kind_type is (complete_within, start_within,
  complete_path_within, complete_start_within);

type DIS_t_constraint_type is
  record
    t_constraint_kind          : DIS_t_constraint_kind_type;
    preference_value           : DIS_preference_range_type;
    time_value                 : DIS_time_type;
    software_id_list           : DIS_softw_id_list;
    parent_mapping_constraint  : DIS_mapping_constraint_type;
    next_t_constraint          : DIS_t_constraint_type;
    previous_t_constraint      : DIS_t_constraint_type;
  end record;

type DIS_p_constraint_kind_type is (place_together, place_separate,
  place_at);

type DIS_p_constraint_type is
  record
    p_constraint_kind          : DIS_p_constraint_kind_type;
    preference_value           : DIS_preference_range_type;
    hardw_id                   : DIS_hardw_id_type;
    softw_id_list              : DIS_softw_id_list;
    parent_mapping_constraint  : DIS_mapping_constraint_type;
    next_p_constraint          : DIS_p_constraint_type;
    previous_p_constraint      : DIS_p_constraint_type;
  end record;

```

Figure 12: Mapping Constraints Data Declarations

A task assignment is the result of running an allocation algorithm on a set of software and hardware nodes with a set of timing and placement constraints.

4. Future Directions and Conclusions

The development of DIS is in the first year of an on going effort. There are several tasks that are planned for future development. These are elaborated below.

1. Library of services

In addition to setting up standards for data structures, DIS should also provide operations such as load/unload, add, delete, update and queries.

2. Importing results to design capture
Provide feedback functions to incorporate the results of the optimization back into the design without modifying the structural components.
3. Exporting results from Simulation/Optimization systems
Interface specifications components can represent results from simulation/optimization systems. This should not only enhance the process, but also provide a standard data representation for developing extract functions for the target systems.
4. Alternative representation languages
Use of high level object-oriented languages to represent DIS has the advantage of inheritance, allowing a higher degree of reusability.

Additionally, now that several versions of DIS have been produced and, as it becomes more robust, participation in various standards organizations will begin. Involvement in the CASE Document Interchange Format Working Group is expected to begin by third quarter of 1992.

References

- [BlSp] Bladen, J.B., D. Spenhoff, "Ada Semantic Interface Specification (ASIS)", Proceedings of Tri-Ada '91, pp. 6-15, October 1991.
- [BOOC] Booch, G., "Object Oriented Design with Applications", The Benjamin Cummings Publishing Company, Redwood City, CA, 1991.
- [CCCC] Computer Command and Control Company, "Software Engineering Environment for Parallel/Distributed Systems", Final Report, Contract #N6092189-C0127, October, 1990.
- [Davi] Davis, L. Ed., Handbook of Genetic Algorithms, Van Nostrand Reinhold, 1991.
- [Gold] Goldberg, D.E., Genetic Algorithms in Search Optimization and Machine Learning, Addison Wesley, 1989.
- [Hoan] Hoang, N.D., "The Essential Views Of Systems Development", Proc. 1st Systems Design Synthesis Technology Workshop, September, 1991.
- [HoHN] Howell, S., Hwang, P., Nguyen, C., "Expert Design Advisor", Proc. 5th Jerusalem Conference On Information Technology, IEEE Computer Society Press, Los Alamitos, CA, October 1990, pp. 743-756.
- [HoNH] Howell, S., C. Nguyen and P. Hwang, "Design Structuring and Allocation Optimization", Proc. Hawaii International Real-Time Systems Conference, January 1992
- [Kare] Karengelen, N., "Multi-Domain Real-Time System Design, Capture and Analysis", Proc. 1st System Design Synthesis Technology Workshop, September 1991.
- [KiGV] Kirkpatrick, S., C.D. Gelatt, and M.P. Vecchi, "Optimization by Simulated Annealing", Science, May 13, 1983, Volume 220, No. 4598.

- [MoMW] Molini, J.J., S.K. Miason and P.H. Watson, "Real-Time System Scenarios", Proc. 11th Real-Time Systems Symposium, IEEE Computer Society Press, Los Alamitos, CA, December 1990, pp. 214-225.
- [NEST] Nestor, J. R., Newcomer, J. M., Giannini, P. and Stone, D. L., "IDL: The Language and Its Implementation", Prentice Hall, 1990.
- [Pear] Pearl, J., Heuristics: Intelligent Search Strategies For Computer Problem Solving, Addison-Wesley Publishing Company, Inc., Reading, MA, 1984.
- [StSh] Staples, G., D. Shavon, "Earthquake Insurance: One Integration Approach", Software Magazine, pp 41-44, February 1992.
- [WaMe] Ward, P., and S. Mellor, Structured Design, Yourdon Press, Englewood Cliffs, NJ, 1972, 2nd ed.

Integrated System Evaluation

*Dong Kil Choi
Naval Surface Warfare Center-Dahlgren Division
Silver Spring, MD 20903-5000
(301) 394-1382*

Abstract

This paper will address issues related to the integration of the systems evaluation methodologies including background, problems, and possible solutions. As systems and their development become more complex, it becomes critical that evaluations and assessments of the system's behavior become an integral part of the system development life cycle. An integration of the evaluation methods into the design capture will allow more of the assessments to become a part of the design decisions of the system.

Introduction:

Current state-of-the-practice of the modeling and analysis methodologies provides fragmented use of the capabilities that are available to the development cycle. Modeling and evaluation capabilities are not as closely tied to the core of the design cycle as needed to provide a seamless path between design synthesis and system evaluation. A designer who wants to test a certain part of the system (or some aspect of the whole system) resorts to the evaluation capabilities to make an assessment of the system. To date, however, there are no formal techniques to apply the information that is obtained from the assessment to the design specification [CHO91]. There needs to be a better information flow between design capture and performance evaluation models. Although the narrow definition of performance does not include reliability, availability, and security, this paper addresses these issues.

Currently, there is a large gap between the representations of design capture and performance evaluation models, although there has been much research that has tried to bridge this gap (such as Teamwork and ADAS, STP and SES Workbench). There are standards efforts like CDIF that are addressing the interchangeability among CASE tools that may serve as an intermediate format between CASE representations and performance evaluation models. These efforts are immature and do not address semantic issues for the most part.

One of the many problems with not having integrated capabilities in complex system development stems from the fact that typically several contractors work on large projects. Not all of these contractors will use compatible tools or methods. This type of a problem usually results in inconsistent design specifications and incoherent system designs.

Within the performance evaluation environment, the size of the system under development (SUD) becomes a quick limiting factor. The number of states in the state dependent representation will typically increase at an exponential rate to the number of nodes. There have been many proposals that have tried to address this issue starting from having a hierarchical representation of a behavior to a partitioning of the SUD. Another possible solution may be to apply a sensitivity analysis early in the development cycle to identify system parameters that greatly affect the performance with even the slightest change. In effect, a sensitivity analysis identifies critical areas of the system that deserve a more detailed evaluation.

Large complex computer-based systems are typically characterized as discrete event dynamic systems (DEDS). Since there is no single paradigm for DEDS, each of the modeling techniques offers advantages to certain types of analyses, but not to all. Just as a design specification needs multiple views to provide a comprehensive description of the system, a performance evaluation needs to use multiple models to provide a comprehensive assessment of the system. For example, finite state machines may be suitable for some communication protocol specification, but are not suitable when unbounded queues are present. Some claim that a Petri-net can be used to represent any system. While this may be true, one can use other modeling approach to represent any system, although the representation may be so complex that it is incomprehensible to everyone except to that designer. The suitability of the modeling techniques to perform certain assessment is the salient requirement, making maximum use of the inherent characteristics of the modeling technique. The ease of use (of representation and analysis) and available analysis capabilities may make it more beneficial to use one modeling approach versus another. Therefore, it is apparent that to make optimum use of the evaluation capabilities, one needs to employ more than one model to represent and evaluate the SUD [GOE91].

The following sections discuss some of the ways that current research efforts have tried to address the problems and issues discussed above. The first section discusses the use of multiple models and their integration into the design process. The second section discusses the integration of performance evaluation models and design capture representations. The third section discusses the long-term goal of unifying the design capture, performance models, and other specifications (such as requirements, implementations, etc.).

Transformation Among Models:

DEDS do not have a single paradigm to represent the system. Each model that has been developed has certain advantages and disadvantages. A systems engineer, however, needs information about the behavior of the system that typically requires more than a single model.

One near-term solution to the current limitations of a single model may be to use multiple models as multiple "views" of the system--just as design capture uses multiple views to represent the design of the system. Then, as is the case in design capture, traceability and a consistency become salient and critical features.

Each "view" has a set of evaluation criteria or metrics that it determines when evaluation methods are applied to the representation. For example, semi-Markov reward models can be used to perform reliability and availability evaluation of the system [TRI91] or a stochastic petri-net to evaluate the dynamics, deadlock, livelock, and reachability of the system. Each "view" needs to capture just enough information about the system to perform its assessments; hence, some of the overhead of carrying unnecessary information is reduced. Within this framework, each "view" has to be consistent with other "views." A value of the parameter in one "view" must be the same in the other "view," or an architecture in one model must be the same as that of the others. To provide consistency, information of one model must be able to be traceable to that of the other. This is a difficult problem because the appearance of the information may differ greatly among models (single parameters in one model may split and distribute across the other models).

The first problem within the transformation among models is that, even within performance models, there are differences in the level of information detail. Some models are used to evaluate the steady state behavior of a system and queuing characteristics, while others simulate the system performing functions. Some of the differences in the level of information are due to the definition of the system by the user and others are due to the limitations of the modeling scheme itself.

The second problem with the transformation among the models is that it is typically not a one-to-one mapping. Therefore, when one model is transformed to another model and is transformed back to the original modeling technique without modifications, this transformed model may look totally different from the original model although be mathematically equivalent. There are also semantic differences among the performance models that make the transformations difficult.

A long-term solution to maximizing the use of the performance evaluation models may be to develop an

intermediate form of representation of the performance models (which is similar in theory to the intermediate form of CASE representation). This intermediate form may just be a way to collect enough data for the transformation into many performance models, or it may be an extension of any one of the modeling techniques. One danger of having a single representation is that the evaluative and expressive powers of a modeling technique may be lost at the expense of making extensions to the existing model. Another danger is information explosion may occur by trying to add so much into this intermediate form.

Transformation Between Capture Methods and Performance Evaluation Models:

A transformation technique between a design capture representation and a performance evaluation model attempts to use the performance model as a design specification as well as a guide in the design. Many design decisions are made in the evaluation and assessment stages of the system development life cycle, therefore, there needs to be a closer linkage between a design capture and performance model representations. A bi-directional transformation between design capture and performance model would provide a means to validate the evaluation results.

Our work in this area attempts to bridge the gap among the currently developing multi-paradigm views of the system [HOA91], including a resource library, a resource allocation and optimization tool [NGU91], and an intermediate form of a performance model. Problems that occur due to differences in the representation include semantic inconsistencies, lack of information, and ambiguities.

Like most new tools and methodologies, design capture had a basic structure, and additions were made to those structures to answer other questions or to address inadequacies (such as real-time extension). The objective of design specification tools, however, differs from that of evaluation tools and therefore information that is contained in the design capture differs from that of performance evaluation models. As such, there is some information that is not provided in the current design capture, such as a resource model. This missing information must be added to transform the design capture to performance evaluation models.

The second problem is also caused by the different objectives of these two representations. For example, in the data flow diagram, the functional decomposition in the design capture stops when a systems engineer can map a function to a resource (whether it be hardware, software, humanware, or a combination). However, even at the lowest level of the functional decompositions, there may be a lot of ambiguities due to multiple inputs and outputs of a single process bubble and a sequence of interactions of the processes. These ambiguities must be removed for many modeling methods.

The flow of information from the design capture to a performance model is difficult. But having the information flow in the other direction, from performance evaluation model to design capture, is more difficult. This task, however, may be more crucial for the widespread use of the systems engineering tools. Once an evaluation is performed, information obtained from the evaluation needs to be fed back into the design capture. This information is pertinent to the decisions made at the design capture, yet there is a lack of formal methods to feed back this information. There may also be design decisions made on the system during the evaluation phase, on the performance model. These decisions also have to be reflected back on the design specifications. This transformation also validates that the design matches the evaluation.

An example of the problem that may arise in a direct transformation from performance evaluation models to design capture is separation of hardware, software, and functionality. When the design capture is transformed to performance models, some of the functions are performed by the hardware, some are mapped to software that gets mapped to the hardware, and some are mapped to humanware. Differentiating the changes in the performance model (into what actually is a function and what is just a resource) may not be done automatically.

There are three goals which address this issue of having a bi-directional flow of information: short-term, mid-term, and long-term. A short-term goal is to attach a note of the changes and pass it to a CASE representation. The decisions on making changes then fall on the design capture side. A mid-term goal is to semi-automate the transformation. Someone working with a tool like DESTINATION [NGU91], which is used

to perform resource allocations and optimization, has an access to both design capture representations and performance evaluation models. Once the changes are made to the performance models, DESTINATION resource candidate allocation may be used to map the changes. These changes may be as simple as the reallocation of existing resources. In this case, changes will be identified in the resource candidate allocation, and they may be made in the DESTINATION in a semi-automated fashion. The long-term goal is, of course, to automate the process.

Global Representation ?:

A global representation is not something that can be obtained today or tomorrow as much as it is a goal (or a focal point) to which the industry is moving. Whether this goal is achievable is questionable [ZAV91]. At this moment, there is a lack of a seamless process from requirements specification to design, evaluation, and the implementation phases of the system development cycle. A global representation could serve as a requirements specification and, if it is robust, could serve as a design specification.

At this time, there is research on forming a standard CASE data interchange format and some research on developing pseudo-standard performance models. Convergence of these representations may be the unifying representation--harbingers of the global representation. However, these representation capture only a small amount of the information needed.

The first of many problems with forming a global representation is balancing the amount of information. The objective would be to have just enough information to specify the system, be able to perform analysis, and use it to implement the system; otherwise, redundancies and information overload would arise.

The second problem is maintaining traceability of the information. The global representation of a system contains more information than needed to perform certain functions (such as timing analysis); therefore, a transformation would be used to perform these functions. After these functions are performed, any changes made must be reflected back to the global representation. Although the changes to the global representation would be guaranteed by the correctness of the transformation, there needs to be a way to trace the changes in the global representation to insure that other aspects of the system are not affected by those changes.

Conclusions:

The issues that are addressed in this paper are not by any means complete. They do represent some of the major issues that have been encountered in this on-going research effort.

Acknowledgements:

The author would like to acknowledge the technical contributions of Steve Howell, Cuong Nguyen, Ngocdung T. Hoang, Michael Jenkins, and Michael Edwards. Their feedback and comments have contributed greatly to the development of this paper. The author would also like to thank ONT, our sponsor, and Adrien Meskin from Advanced Technology and Research Corporation for her editorial support.

References

- [CHO91] D. Choi, J. Youngblood, P. Hwang. Modeling Technology for Dynamic Systems. In *Proceedings of the 1991 Systems Evaluation and Assessment Technology Workshop*. 1991, 89-94
- [GOE91] R. Goettge, E. Brehm. Integrated System Modeling and Assessment. In *Proceedings of the 1991 Systems Evaluation and Assessment Technology Workshop*. 1991, 31-34
- [HOA91] N.D. Hoang. The Essential Views of Systems Development. In *Proceedings of the 1991 Systems Design Synthesis Technology Workshop*. 1991, 3-9
- [NGU91] C. Nguyen, S. Howell, P Hwang. System Design Structuring and Allocation Optimization. In *Proceedings of the 1991 Systems Design Synthesis Technology Workshop*. 1991, 117-128
- [TRI91] K. Trivedi, J. Muppala, S. Woollet, B. Haverkort. Composite Performance and Dependability Analysis. In *Proceedings of the 1991 Systems Design Synthesis Technology Workshop*. 1991, 299-328
- [ZAV91] P. Zave. An Insider's Evaluation of PAISLey. *IEEE Transactions on Software Engineering*, Vol. 17, No.3, March 1991

ORGANIZING TOP-LEVEL SYSTEM REQUIREMENTS

Ralph D. Jeffords
Locus, Inc.
Alexandria, VA 22303

Bruce G. Labaw
Naval Research Laboratory
Washington, DC 20375

1. INTRODUCTION

Top-level system requirements are analogous to software requirements in many respects. The purpose of each is to describe "what" is required for a system, subject to constraints, without giving details of "how" this should be accomplished. Historically the methods and tools for software requirements specification have been adopted as also suitable to systems requirements [Dorfman 1990].

For a small, single computer-based system there may be little distinction between the system and software requirements. However, for large complex systems the software comprises just some of the components of what may be a distributed system with many hardware, software, and human-oriented subsystems. Ideally the system requirements should be understandable both by the customers (as C-requirements) and by the system developers (as D-requirements) [Rombach 1990]. Developers require a precise statement of requirements that can be verified, i.e., there must be some cost-effective procedure for determining whether an implementation satisfies the requirements. On the other hand, such a precise statement may require concepts and notations that are unfamiliar to the customers. More realistically, one must be able at least to validate the requirements in some manner with respect to the customer's needs.

The SCR (Software Cost Reduction) methodology attempts to provide a basis for customer as well as developer understanding of software requirements by the use of semi-formal representations and a well-defined set of principles. The application of these principles is demonstrated in a complete example of the software requirements of an actual Navy system: the operational flight program of the A-7E aircraft [Heninger 1980, Alspaugh et al. 1992]. The formal representations are largely based upon finite state machine models for representing the system behavior. The SCR methodology emphasizes an external "black-box" view of the system without any premature, design-level partitioning of the system.

In light of these objectives of the SCR methodology, we propose extensions to that methodology to handle system requirements rather than just software requirements. These extensions are based in part on [van Schouwen 1990] and [Rose et al. 1991] with additional insight from [Clements et al. 1992] and [Hester et al. 1981].

We first introduce some general principles that guide our development of a systems requirements document. Next, we outline how these principles can be used to organize top-level systems requirements following the pattern of the SCR methodology. Details of much of this organization are yet to be determined; much of our focus in the discussion below is on the issues to be resolved.

2. GENERAL PRINCIPLES

- The specification should anticipate likely system changes (e.g., changing requirements, initial subsets to be expanded later, changing hardware characteristics, etc.). To facilitate this, the emphasis should be on specifying a family of related systems rather than a single system. Likely changes should be clearly documented so that the succeeding system design can be tailored to make those changes easy.
- The requirements specification should be organized to separate concerns. For example, likely system changes should be separated from the rest of the specification. Many forms of abstraction also promote separation of concerns as well as ease of change, e.g. symbolic representation of data values to separate concerns of representation and usage within the system.
- Development of the specification should focus on the questions that need answering before formulating the answers. Rather than prematurely giving an answer, it is appropriate to record each unresolved issue with some convention such as "TBD."
- To control redundancy, each entity should be defined in a single place (whether this be in the specification per se, or at a higher level as part of a specification generator). As necessary there should be automated control (e.g., macros) for requisite multiple copies of such entities to preserve consistency. Examples from the SCR methodology are data types, system generation parameters, and terms.
- The requirements specification should be a reference document; i.e. the emphasis should be on finding specific information rather than giving a general overview of the system. A general overview may be included in this document or furnished separately. To facilitate use as a reference document, various indices and cross references (or their automated equivalent in, e.g., hypertext) must be included.
- Care should be taken that only requirements are included in the requirements document. Premature design decisions must be avoided. Required design constraints should be clearly marked.
- The requirements should be stated as formally as possible since formal notations are more likely to be consistent, unambiguous, and concise.

3. TOWARDS A REQUIREMENTS STRUCTURE

We consider three broad areas of system requirements [STARTS 1987]:

- Functional requirements describe "what" behavior is required of the system.
- Nonfunctional requirements are attributes of the system not covered by the functional requirements.
- System development requirements address the process by which the system is developed and evolves over its lifetime.

The functionality of the system should be the focus in organizing system requirements [STARTS 1987]. Nonfunctional requirements and system development requirements should not be treated as second-class citizens but should complement the major structure provided by the functional requirements. The mechanism for implementing this structure is left open. Requirements specifications stored within a database allow for flexibility in that many different groupings of data are available. For exposition we shall consider a division into chapters and sections with the understanding that each chapter or section represents a logical view of the overall requirements database.

Chapter 1: INTRODUCTION

The introduction includes such information as organizing principles, notations, a brief overview of the system, and synopses of the remaining chapters.

Chapter 2: FUNCTIONAL REQUIREMENTS

We partition the system under development into the environment within which the system will function, and the system itself. The Environmental State Variables represent the interface to the environment. The System Modes provide the control model for the behavior of the system as a simplification of the overall state of the environment. Finally the System Functions describe the actual behaviors of the system as related to Environmental State Variables and the System Modes.

Section 2.1: Environmental State Variables

Environmental (state) variables [van Schouwen 1990, Parnas and Madey 1991] are defined in this chapter. These time-varying variables model the external environment of the system. They include physical quantities (e.g., temperature and pressure), readouts of displays, and even human user characteristics (e.g., typing speed of operators external to a system). Each variable is either a monitored variable, which is measured as input to the system, or a controlled variable, which is a quantity that must be controlled by the system, or a variable may be both monitored and controlled. The environmental variables express the interface to the system as well as additional relevant factors in the environment. It is important to include environmental restrictions (e.g., physical laws) in order that the environmental model sufficiently reflects reality.

ISSUES:

- How extensive must the environmental model (captured by the environmental variables and restrictions) be? It must capture those aspects relevant to the system being specified; the restrictions should rule out system states that are impossible. It is perhaps better to overspecify the environment than to underspecify it.
- What is the best way to structure the environmental variables? For which types of systems may object-oriented techniques be useful in structuring the environmental variables?

Section 2.2: System Modes

A mode class represents an equivalence class of the overall system state, i.e., the individual modes of a mode class partition the system—each system state belongs to exactly one mode. Modes are useful in simplifying complex environmental conditions and in capturing useful history of the system evolution. Transitions between modes correspond to the event of one or more environmental variables having changed (or may be defined via conditions of the environment rather than events). Several different mode classes may be useful in simplifying the expression of system requirements. Modes should represent only externally visible aspects of the system state.

ISSUES:

- How does one ensure that only externally visible behavior is captured via modes? One test for this is that the modes of a mode class only capture information on environmental variables even if those environmental variables are not explicitly specified.
- What is the best way to represent a mode class and its transitions? Both tabular notations (e.g., SCR tables [Alspaugh et al 1992, van Schouwen 1990]) and graphical notations (e.g., Statecharts [Harel 1987] and Modechart [Jahanian et al. 1988]) should be considered; each has its advantages and disadvantages for ease of change, understandability, etc.

Section 2.3: Descriptions of System Functions

All system functions should be described in this chapter. Each system function should be a description of the external behavior of some aspect of the system in terms of the state of the environment. In describing each system function, one must include the relevant modes during which that function is applicable. We include both functions during normal operation and error situations, since it may be difficult to distinguish these, or there may be several levels of degraded performance that must be specified.

ISSUES:

- What is the best way to structure the various functions? Do we want to structure as normal versus abnormal operation or use some other criterion?
- What is the best way of relating nonfunctional attributes to system functions? The recommendation in [van Schouwen 1990] is that the functionality should be focused upon an ideal model of the system with timing, accuracy, and other non-functional characteristics introducing the allowable tolerances to this ideal model. We believe that such an ideal system model abstracts out too much of the requisite behavior of the system, e.g., timing and accuracy should be intrinsic parts of the system's functional behavior.

Chapter 3: NONFUNCTIONAL REQUIREMENTS

We consider two areas of the nonfunctional requirements:

- Design constraints express "how" functional requirements must be refined
- Additional nonfunctional constraints address other aspects of the system

Section 3.1: Design Constraints

Design constraints include descriptions of the major system subcomponents whose functionality has been specified to a level of detail that is more appropriate to the system design, i.e., there is some "white box" view of the functionality [Miller 1984]. These constrained subcomponents include hardware, software, and human interaction within the system - most commonly human-computer interaction. In a general systems setting it must be clarified which human interactions are external to the system, and which are internal to the system.

This chapter should basically include what would be found in the system design documents for those components, only moved to the requirements document and clearly labeled as design constraints. The detailed design information here should not take the place of the requirements listed elsewhere in the document, but should be a refinement of those requirements; in other words these constraints should be removable without affecting the rest of the document.

ISSUES:

- Why not let a design constraint replace the more abstract requirement, which the design constraint refines? It may be tempting to take this shortcut in order to shorten the requirements document. This is not to be recommended since it muddles an abstract view of the overall system due to the level of detail, and it makes changes to design constraints more difficult.

Section 3.2: Other Nonfunctional Requirements

Additional nonfunctional requirements cover a broad range for general systems. We identify three categories here as examples:

- Interface constraints: hardware interfaces, software interfaces, man-machine interfaces (MMI's).
- Dependability constraints: safety, security, stochastic performance, deterministic performance, reliability, availability, accuracy, maintainability.
- Physical requirements: dimensional limits, power consumption, environmental conditions (weather, noise, radiation), climate control, etc.

As with functional requirements, care should be taken that these other non-functional requirements do not overconstrain the specification to rule out what might be acceptable designs and implementations. For example, fault-tolerant computer hardware should be considered as one possible system design in achieving an overall system reliability.

ISSUES:

- How should nonfunctional requirements be organized? The many different forms of nonfunctional constraints in a general system differ widely in scope and interdependence.
- To what extent can various categories of nonfunctional requirements be expressed formally? Is it useful to have at least a formal syntax or template for expressing such requirements (or would that be too restrictive)?

Chapter 4: SYSTEM DEVELOPMENT REQUIREMENTS

In building large complex systems the customer may also require specific methods, tools, and procedures to be followed to ensure that the system development process is under control.

Section 4.1: Required Subsets

Required subsets (e.g., phased deliverables) and variants of a system are documented here. Appropriate information that may differ among subsets or variants may include timing requirements, hardware requirements, physical requirements, etc.

Section 4.2: Expected Types of Changes

Special care should go into documenting change within a system. It is important to record fundamental assumptions (i.e., those aspects and decisions that will never change over any system meeting the requirements) as well as changeable assumptions. These two types of assumptions apply to all areas of the requirements specification.

Expanding upon these assumptions, there should also be rationale, as appropriate, gathered during any analysis that preceded the establishment of the system requirements. Rationale is especially important for system design constraints.

ISSUES:

- Should fundamental and changeable assumptions be grouped together? It is probably clearer to separate them.
- Are fundamental unchangeable assumptions necessary? One view is that anything not explicitly stated as changeable is implicitly unchangeable. We disagree with this view since implicit assumptions are likely to be ambiguous.
- How much rationale should be included here (or in companion documents)? Some rationale is necessary as a check that the recorded assumptions are valid.

Section 4.3: Other System Development Requirements

Other requirements in this category include life cycle concerns (e.g., testing requirements, documentation standards), installation procedures, project management, and quality assurance [STARTS 1987].

ISSUES:

- Similar to Other Nonfunctional Requirements, it is difficult to formalize process-oriented requirements.

Chapter 5: GLOSSARY OF TERMS

This glossary covers all terms, jargon, abbreviations, conventions, etc. that are normally used within the general domain of the system (e.g., the avionics domain). The main purpose is to provide the specifiers with the requisite domain terminology for communication with the customer. If appropriate, reference to standard glossaries of domain terminology (cited in Sources of Additional Information) may replace part of this chapter.

Additionally special terms (bracketed by !---! in A-7E requirements) specific to this particular project should be defined. These special terms provide a mechanism for consistently recording and using names that might otherwise be confused with informal usage. Term definitions may range from informal natural language descriptions to formal descriptions such as macros, mnemonics for system parameters, or abbreviations for complex mode expressions. In any case a term is useful for hiding details of a concept, i.e., separating the concern of where and how that concept is used versus its detailed definition.

The terms defined here should be used consistently throughout the rest of the document to aid in customer understanding as well as conciseness. For example, relevant terms should use the same acronyms as those defined for the domain.

ISSUES:

- What is the best way to organize the different sorts of items in the glossary?

Chapter 6: SOURCES OF ADDITIONAL INFORMATION

This section gives references to all relevant publications related to the system specification (e.g., computer manuals, appropriate standards, detailed hardware descriptions). It also records names, addresses, phone numbers, etc. of all personnel involved as either customers or systems requirements specifiers indicating their role or expertise related to the requirements specification.

4. ACKNOWLEDGMENTS

We would like to thank Connie Heitmeyer and Mike Edwards for useful suggestions in improving this paper.

5. REFERENCES

STARTS 1987.

U. K. Dept. of Trade and Industry, *The STARTS Guide, Vol. 1 (2nd ed.)*, National Computing Centre, Manchester, UNITED KINGDOM, 1987. (Excerpt reprinted in Dorfman and Thayer, *Standards, Guidelines, and Examples on System and Software Requirements Engineering*, pp. 320-367.)

Alspaugh et al. 1992.

T. A. Alspaugh, S. R. Faulk, K. H. Britton, R. A. Parker, D. L. Parnas, and J. E. Shore, "Software requirements for the A-7E aircraft (Release 3)," NRL Report 9194, Naval Research Laboratory, Washington, DC, (to appear 1992).

Clements et al. 1992.

P. C. Clements, C. E. Gasarch, and R. D. Jeffords, "Evaluation criteria for real-time specification languages," NRL Memorandum Report 6935, Naval Research Lab, Washington, DC, Feb. 1992.

Dorfman 1990.

M. Dorfman, "System and software requirements engineering," in *System and Software Requirements Engineering*, R. H. Thayer and M. Dorfman, ed., IEEE Computer Society

Press, Washington, DC, pp. 4-16, 1990.

Harel 1987.

D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming* 8 (3), pp. 231-274, June 1987.

Heninger 1980.

K. L. Heninger, "Specifying software requirements for complex systems new techniques and their application," *IEEE Trans. Softw. Eng.* SE-6 (1), pp. 2-13, Jan. 1980.

Hester et al. 1981.

S. D. Hester, D. L. Parnas, and D. F. Utter, "Using documentation as a software design medium," *Bell System Tech. J.* 60 (8), pp. 1941-1977, Oct. 1981.

Jahanian et al. 1988.

F. Jahanian, R. S. Lee, and A. K. Mok, "Semantics of Modechart in Real-Time Logic," in *Proc. 21st Hawaii Intl. Conf. on System Sciences*, Jan. 1988.

Miller 1984.

E. F. Miller, Jr., "Software testing technology: an overview," in *Handbook of Software Engineering*, C. R. Vick and C. V. Ramamoorthy, ed., Van Nostrand Reinhold, New York, NY, pp. 359-379, 1984.

Parnas and Madey 1991.

D. L. Parnas and J. Madey, "Functional documentation for computer systems engineering (Version 2)," CRL Report 237, Computer Research Lab, Telecommunications Research Inst. of Ontario (TRIO), McMasters Univ., Hamilton, Ontario, CANADA, Sept. 1991.

Rombach 1990.

H. D. Rombach, "Software specifications: a framework," SEI Curriculum Module SEI-CM-11-2.1, Softw. Eng. Inst., Pittsburgh, PA, Jan. 1990. (Reprinted in Dorfman and Thayer, *Standards, Guidelines, and Examples on System and Software Requirements Engineering*, pp. 368-407.)

Rose et al. 1991.

A. Rose, C. Heitmeyer, and B. Labaw, "Languages for requirements specification: experience with a tabular approach," in *Proc. 1991 Systems Design Synthesis Technology Workshop*, Naval Surface Warfare Center, Silver Spring, MD, pp. 107-113, Sep. 10-13, 1991.

van Schouwen 1990.

A. J. van Schouwen, "The A-7 requirements model: re-examination for real-time systems and an application to monitoring systems." Tech. Report 90-276, Queen's Univ., Kingston, Ontario, May 1990.

SYSTEM ENGINEERING II

THE COMPONENT MANAGER

Supporting Interactive and Automated Retrieval of Real-Time Software Components ¹

W. Rossak, A. Stoyenko, and L. R. Welch.
Real-Time Computation Laboratory and
Systems Integration Laboratory
Department of Computer and Information Science
New Jersey Institute of Technology
University Heights
Newark, NJ 07102
Telephone: (201) 596-6597
E-mail: rossak@pluto.njit.edu
Fax: (201) 596-5777

Topics: Software Reuse for Real-Time Applications,
Interconnected Systems, Modeling Technology

April 27, 1992

¹Supported in part by the State of New Jersey, and by the Instrumentation for Real-Time Applications Grant from AT&T

Abstract

The development and maintenance of real-time systems has become more and more important. However, the tool support for reuse of real-time components is by far not as elaborated and conceptually well defined as this would be necessary. We present an approach that combines existing technology in software reuse, formal specification, real-time schedulability analysis, and advanced matching techniques into a comprehensive framework for reuse of real-time software. The basic concept is the reuse of software components which are stored in a highly structured library system. A typical component exports a type and operations to be used on variables of that type. Components/modules may be generic (i.e., parameterized by types, by operations, or even by other modules). To be used, modules must be instantiated. This structural element of the Component Manager is based on experience with two prototypes, the Software Archive [11, 9, 12] and the RESOLVE specification language [5, 2, 4]. To support the formal specification of real-time software, RESOLVE is extended with special constructs to express information regarding timing, periodicity, etc. Within this context, we consider a mix of user-guided and automated retrieval/classification achieved by structural support and formal specification. The results of this "local" matching effort are used to conduct an evaluation ("global" matching) by running an analysis of how the target system will react, with regard to its timing properties, to a reuse of the found component. This paper gives an overview of the static structure used to specify the Component Manager. This part of our work is based on the Software Archive concepts. It also discusses extended RESOLVE, a language for the specification of reusable real-time components and systems. Additionally, it presents a basic algorithm for (automated) retrieval of components in the Component Manager and presents the concepts of global matching with existing systems.

keywords: software reuse, component libraries, formal specification for real-time, schedulability analysis, global and local matching.

1 Introduction

Software reuse is one of the main factors in increasing software productivity. However, while we can see a wide variety of approved concepts and even a few successful practical applications in the area of business and information systems [19], we still have problems in application domains where performance, timing behavior, and other non-functional attributes are essential.

Our goal is to develop a reuse support tools that addresses these problems, especially the needs and requirements of real-time systems development. This paper presents the conceptual framework for this approach, discusses existing prototypes and techniques, and integrates them into a comprehensive tool environment, the Component Manager.

There are four main elements in this framework that characterize our approach:

- A highly structured library of components.
- A hybrid mix of interactive and automated retrieval.
- Special support for formal specification of real-time characteristics.
- An evaluation of impacts on the target system.

The Component Manager is based on the well-known and successful concept of a library of components. However, this library is not flat, providing an unstructured set of component descriptions, but relies heavily on an explicitly defined classification structure. This structure is not only visible to the user but it is also specified and maintained by him. This static structure is the basis for all other elements of the Component Manager and allows one to mix relatively informal concepts with formal specifications.

Using both aspects, informal and formal elements, retrieval and classification utilize the structure to provide the user with a highly flexible interface. It is open to the users to decide if they want to use the system in a user-driven, interactive browser-mode, or if they want to hand over to automated matching algorithms that take a formal specification as input and match it to a formal specification in the library. What is common to both alternatives is that they rely on the predefined static structure to narrow the search space instead of trying to match with all components in the library.

By mixing the informal and the formal model for reuse libraries [13], we achieve a situation where the user can guide the system interactively by navigating in the structure of the Component Manager without losing the advantage of machine support to handle bulk data. This hybrid approach allows one a better fine tuning and guidance of the system functionality and is a possible solution to the problem of inflexible algorithms that are targeted only towards one aspect of the component description and can not handle other attributes, e.g. timing specifications.

To support this type of system, every component description must have a formal and an informal part. The informal part is mainly used during the browsing process and takes the form of a multipurpose component description that can be used in different classification structures (see section 2). The formal part supports the automated retrieval, a task that can not be fulfilled by the informal, free format elements. The formal specification also includes special constructs to define the real-time aspects of the component.

These real-time characteristics are not only used to decide if a component fits the needs of a user by comparing component description and needs statement, they are also the major input to evaluate the found component(s) as part of the target system. As opposed to the (local) matching of component characteristics during retrieval in the library, this step recalculates the behavior of the whole system whose execution is influenced by timing and resource needs of the currently evaluated component. This global matching can be seen as an additional evaluation step which, if necessary, reorders the list of "locally" matching components.

Section 2 gives an overview of the static structure used to specify the Component Manager. This part of our work is based on the Software Archive concepts. Section 3 describes extended RESOLVE, a language for the specification of reusable real-time components and systems, augmenting the informal concept of the Component Manager and supporting automated retrieval and classification. Section 4 presents the basic algorithm for retrieval of components in the Component Manager and discusses aspects of the local matching algorithm. The process of global matching, on the basis of incremental schedulability analysis, and its relation to local matching is outlined in section 5.

2 The Basic Structure of the Component Manager

The structure of the Component Manager is derived from a generalized view of the software development process [9], and achieves independence from certain applications or projects. Each Component is described as a unique entity and linked to other components in the library by application independent relationships [10, 12].

A component is not restricted to a representation on the source code level. It may have any form of representation ranging from specification to source code. Furthermore, potentially reusable components are classified according to their level of decomposition, e.g., as systems or sub-systems. Similarity between components is described by using a generalization hierarchy with strict attribute inheritance.

A component is a part of an existing system. It is reviewed for possible reuse and incorporated into the Component Manager by classifying it according

to its structural relations to the other modules in the repository. A module description - as far as it will be described here - consists of the following parts:

- Contents definition.
- Interface and placement/positioning information.

The contents definition includes the original description of the module, as derived during its development process. It is the basis to define attributes which are independent from a notational form used in a specific project. These attributes, classifying the module, are also stored in the contents definition part of the module description. They are of superior importance for determining the location of the module in the repository structure and for manual retrieval operations.

A standardized and generic specification of the component on the level of "concepts" is used to define a normalized component description. These concept level descriptions support easier understanding for the user and can be used during the automated retrieval operations. Related to such a concept description, different realizations can be stored, providing design and implementation alternatives.

The interface and placement description contains all information necessary to determine the place of the module in the repository's structure, thus capturing its relations to the other components. To support this goal, the global structure of the Component Manager includes as its main elements a decomposition dimension and a generalization dimension [10]. The decomposition dimension classifies a module according to its level of system aggregation and is implemented by PART-OF relations. The generalization dimension models similarity between different modules (on one level of decomposition). It is implemented by an ISA relation.

The structure of the Component Manager, with its main factors aggregation/decomposition and generalization/specialization, can be seen as shown in figure 1. There exist different levels of decomposition. Each module is assigned to exactly one of these levels (and may be linked to modules on other levels by means of a PART-OF relation, not shown in the diagram). On each level of decomposition the modules are structured according to their level of abstraction in generalization hierarchies.

The ISA relation is defined in terms proposed in [1]. It is a relatively static form of ISA relation, including strict inheritance of attributes from ancestors but excluding multiple inheritance. The attributes inherited are the classifying attributes placed in the contents definition of the module. The attributes of all related modules on higher levels of generalization are inherited and completed by those derived from the current modules' contents definition.

Different points of view concerning similarity between components and/or different ways to structure the search space can be expressed by the user-view concept [12]. This concept of varying views for different groups of users on

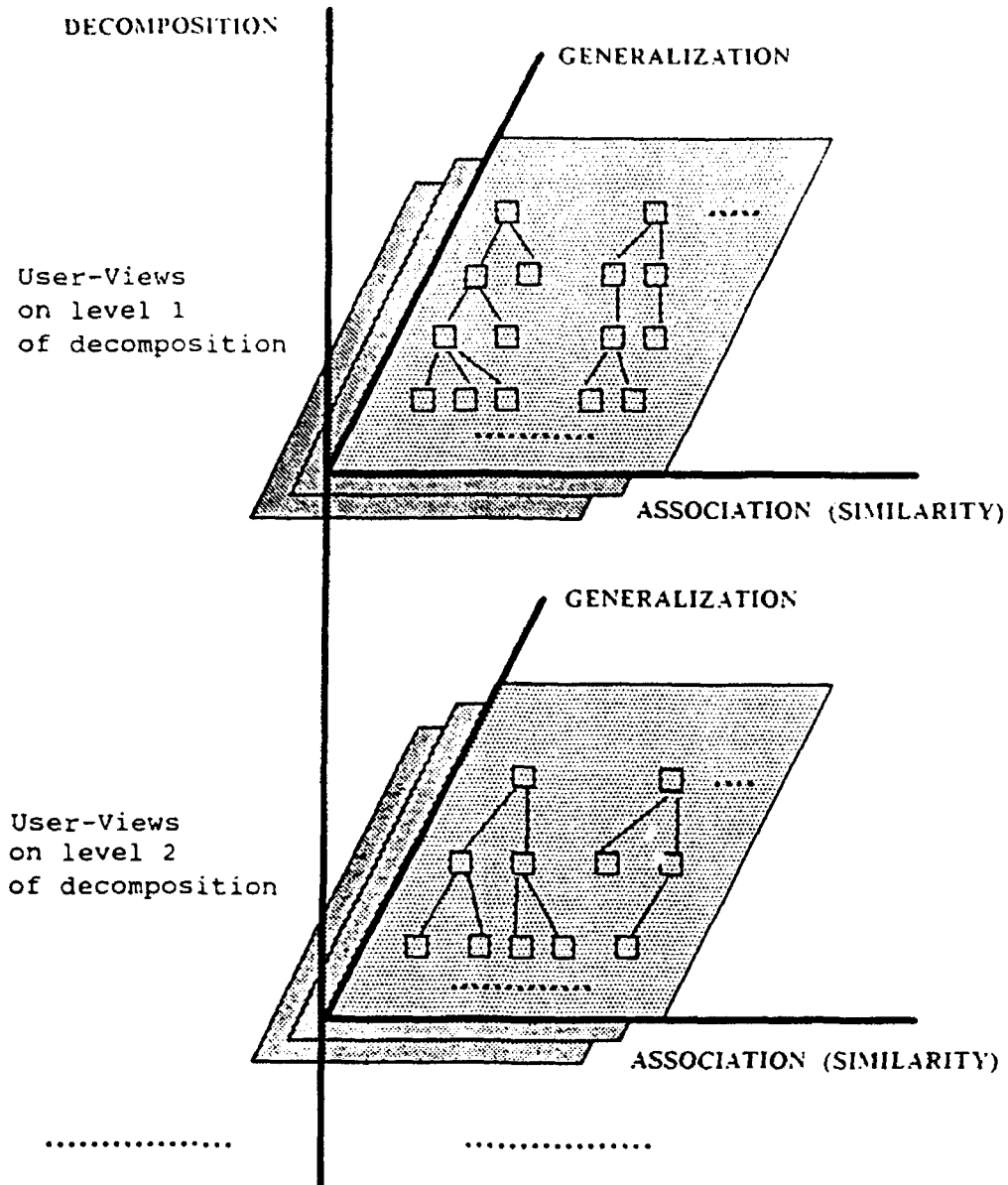


Figure 1: The basic structure of the Component Manager.

the same conceptual structure is comparable to the user-view concept used in database applications.

A user-view consists of components linked by ISA relationships, thus spanning the two dimensions of generalization and similarity-association (see figure 1). Using generalization and attribute inheritance, a class hierarchy of (similar) components is constructed. All components in a user-view must be classified to be at the same level of system decomposition.

If we take into account that the notion of similarity is a semantic concept, which is determined by the point of view of the user, we have to recognize that it will not be adequate to enforce the use of only one such classification scheme on a given level of decomposition. Doing so would make it difficult or even impossible for the user to structure the set of components according to his/her needs.

Therefore, to support modeling of alternative generalizations and similarity classifications, the Component Manager allows the user to build different parallel user-views on one level of decomposition. They provide the means to define different classification structures, reflecting the different points of view of different user groups (such as projects or departments).

This means that a component is classified to belong to exactly one level of decomposition, but may be used in different parallel user-views. The user-views on one level of decomposition represent the set of all specified classifications for the components on that level.

The main concept of the reuse support structure, as presented so far, is to divide the search space (the set of all stored components) into user-views and levels of decomposition. This structure enables a user to retrieve components by browsing through the system by hand or by invoking an automated search support algorithm. While the user can utilize most of the (rather informal) information stored for each component, the algorithm will mainly rely on the standardized formal specification which is part of the components' content definition.

3 The Specification Language

3.1 Basics of RESOLVE

Each specification is called a concept and may have multiple implementations (called realizations). A typical module exports a type and operations to be used on variables of that type. Modules may be generic (i.e., parameterized by types, by functions or operations, or even by other modules). To be used, a module is instantiated by fixing its parameters and choosing one realization. We refer to a module instance as a facility. Each module has an initialization operation that executes when a facility is created, and initializes any state the facility may have. Type initialization and finalization operations must be provided by the

author of a module for each type that the module exports. These operations are required to simplify proofs of correctness of implementations and so that each variable can be automatically initialized (or finalized) upon entry to (exit from) an operation that declares it.

In our work, reusable software components are specified using RESOLVE [4, 20, 5, 2], which is an acronym for Reusable Software Language with Verifiability and Efficiency. In RESOLVE, every program type is modeled using a standard mathematical type and operations on the program type are explained using notations from its mathematical type. Specifications are written using predicate calculus and well known mathematical theories such as integer theory, real number theory, boolean algebra, string theory, and function theory. For example, the type List can be modeled using an ordered mathematical pair of mathematical strings, as shown below.

```
concept List_Template (type Item)
  type List is modeled by
    <left: String (Item),
      right: String (Item)>
  initially, forall L: List, L.left = Lambda and L.right = Lambda

  procedure Reset(alters L: List)
    ensures L.left = Lambda and L.right = #L.left o #L.right

  procedure Advance(alters L: List)
    requires L.right /= Lambda
    ensures thereExists Y: Item, s.t.,
      L.left = #L.left o Y and #L.right = Y o L.right

  function At_Right_End(preserves L: List) returns Boolean
    ensures At_Right_End iff L.right = Lambda

  procedure Insert(alters L: List; consumes X: Item)
    ensures L.left = #L.left and L.right = #X o #L.right

  procedure Remove(alters L: List; produces X: Item)
    requires L.right /= Lambda
    ensures L.left = #L.left and #L.right = X o L.right

  procedure Swap_Right(alters L1, L2: List);
    ensures L1.left = #L1.left and L1.right = #L2.right and
      L2.left = #L2.left and L2.right = #L1.right
end List_Template
```

Different mathematical theories can be used to reason about the behavior and abstract structure of concepts. In each concept, the mathematical model of

the type is stated and the initial value for variables of the type are given. The interfaces of the operations and descriptions of their behaviors follow the type description. Each operation has an optional requires clause, a pre-condition that must be satisfied before the operation is invoked. Each operation also has an ensures clause describing the effect of the operation when the requires clause is satisfied. Ensures clauses refer to the old values of parameters by preceding their names with a pound sign (#).

The type List can be viewed as an ordered mathematical pair of mathematical strings (as shown in an earlier section of this paper). Considering the former model, we see that the list is treated as two segments, left and right. When a client of the List.Template declares a list variable, it is given the initial value described in the initialization section—an empty list. An imaginary cursor points to the position in the list between the left and right portions. Thus, the Insert and Remove operations affect the leftmost item on the right half of the list. The Insert operation is described as the concatenation of an item onto the end of a string. Similarly, the Remove operation deletes the leftmost item of the right string. Advance moves the leftmost item of the right string into the rightmost position of the left string. Reset moves the cursor to the left of the entire list and the operation At.Right.End returns true if and only if the right string is empty. The Swap.Right operation exchanges the right portions of two different lists, permitting implementations that provide efficient swapping of lists.

The RESOLVE specification language provides additional features that did not appear in the specification of the List.Template. Items that can be declared are mathematical constants, variables and functions. These are used only for stating the conceptual view of a module, but typically have counterparts in realizations of the module. Additional information that can be provided includes constraints on behavior of the module, lemmas (to be used in understanding the module and to simplify proofs of correctness), and the initial state of the module. These items are stated as assertions using predicate calculus. The definitions of provided types (in the interface section) can contain such information as constraints on states of variables having the type, lemmas, and initial and final values. Parameters to a concept can be types (as seen in the List.Template), mathematical functions, or other concepts. When a concept is passed as a parameter, any operations and types that it provides, and anything defined in its auxiliary section, can be referenced. To reference operations and types provided by a concept parameter, they must be fully qualified. Additionally, a concept can access anything passed as a parameter to a concept that is a parameter to it. For example, if concept *h* is a parameter to concept *g*, and *g* is a parameter to concept *f*, then assertions in *g* can reference definitions from *h*; and assertions in *f* can reference definitions from *g* and *h*.

To assure that the correct parameters are used when modules are instantiated, restrictions can be stated in the parameters sections of modules. Restrictions can be (1) TYPE.NAME = TYPE.NAME or (2) CONCEPT.NAME = CONCEPT.NAME.

These restrictions enable module designers to prevent incorrect usage and permit static checks to be performed. For example, if a module requires two type parameters, the module designer is permitted to state the constraint that the types must be the same.

To provide hierarchies of components, concepts can enhance other concepts. An enhancement "inherits" the definitions, types, and operations provided by the enhanced concept. Additional operations and types can be provided by the enhancement. For example, below we specify and implement an operation that reverses a list. The reverse operation is secondary—meaning that it is implemented in terms of operations provided by the List_Template.

```
concept List_Reverse_Template enhances List_Template
  procedure Reverse(altern L: List)
    ensures L.left = Lambda and L.right = (#L.left o #L.right)~R
  end List_Reverse_Template
```

RESOLVE is not just a specification language—it also permits multiple realizations of any concept to be written. The RESOLVE implementation language has many noteworthy features. Assignment (copying) of one variable's value to another variable is not a part of the language; instead, swapping the values of two variables is the only built-in data movement primitive. There are no global variables. Instead, operations can access three kinds of data: operation parameters; local variables; and module variables (static variables associated with a particular module instance that are shared among operations exported by that instance). Aliased variables cannot arise, i.e., the data structure representing a variable's value can only be known by one name at any time. No types are built-in; therefore, almost every statement is a call, since every manipulation of a variable whose type is provided by a reusable component is achieved by a call to a facility operation. Modules cannot be instantiated dynamically; i.e., instantiations are declarations that occur outside the code of module operations, and all instantiations are performed when a program begins execution. Furthermore, the types of variables are determined statically, and there are no constructs in the language for expressing parallelism.

3.2 Real-Time Aspects of RESOLVE

Languages such as RESOLVE [5] allow module developers to state the duration interval for operations exported by the modules. While this information is useful for reasoning about the execution time of composite modules, additional language constructs are required to describe systems of real-time processes. We allow system definitions to contain facility declarations, followed by global variable declarations, and then process specifications. A process specification contains facility declarations, variable declarations and code. Additionally, each process description may mention periodicity, deadlines, and external events.

Periodicity refers to the frequency with which repeatedly executed processes are performed. The syntax for denoting a periodic process is:

every x seconds perform f.operation1(a, b, c)

The period of the process is stated in seconds. The behavior of the process is programmed as the operation of a module.

A deadline dictates the maximum time that can elapse from the start of a process invocation until its completion. The deadline of a periodic process is stated as:

every x seconds perform f.operation1(a, b, c) before y seconds elapse

The event-driven process is an important part of many real-time systems. Such a process typically has deadlines. Additionally, there is usually a minimum amount of time that must elapse between occurrences of an event. We allow event-driven processes to be codified as:

on event e perform f.operation1(a, b, c) before y seconds elapse at most every z seconds

Another construct needed is the timer. It allows system developers to insert delays into a processes code so that activities in the controlled environment can be performed before proceeding with function of the process. To insert a delay, a component developer writes "delay x seconds".

Devices are often manipulated by real-time systems. We permit this by encapsulating each device in a module. Thus, device states may be inspected or altered simply by calling a module operation.

4 Retrieval of Components - Local Matching

4.1 The Basic Algorithm

Utilizing the structural layout of the repository and the power of concepts and realizations, the dynamic aspects of the Component Manager include all the necessary utilities and user features to search for an existing module in the repository and to classify a new module.

The software engineer analyzing and refining a given system component is the typical user interacting with the tool. He searches for modules fitting his demands, as derived during the development process, and provides the library with new modules not included in the structure so far [9].

While searching in the Component Manager can be done by everybody who needs an existing system component, it is desirable to allow only a specially trained person to incorporate new components. This role of a "component administrator" can help to keep the system consistent and can avoid violating predefined integrity or organizational rules.

Searching for and classifying components can be seen very much as the same kind of process, only starting at different ends. As the discussion of both processes would go beyond the scope of this paper, only the retrieval is presented in some detail as described in figure 2. (The algorithm presented here is an adapted version of an algorithm presented in [8].)

Starting from given demands, i.e. the specification of a component, the first step is to determine on which level of decomposition the component is most likely to be situated. This level of decomposition can be derived from the level of decomposition the development process is currently working on.

At the determined level of decomposition exist multiple user-views. After picking one of them, perhaps based on a description of the classification strategy used in the user-view, a first approximation of the component searched for has to be located. To do so, look for one of the very general components on top of the generalization hierarchy and pick the one which is most similar to the specification describing the demands. If this component is not a perfect match, a recursive subtree search can start that takes this module as its starting point. This search process will lead to more specialized levels of generalization with more attributes and more information.

The search is oriented towards the ISA relations of the structure and follows a path leading to refined but similar modules (as defined by the ISA-relations). This allows appropriate reasoning about different possible solutions [3] and takes advantage of the structured search space [7]. The quality of the results obtained and the ease of use of the tool rely to a certain degree on the correspondence between the structure of the development process and the structure of the Component Manager.

Many of the functions incorporated in the search process described above can be optimized if they are interactively guided by the user. For example, to determine if a given module is matching or not can be done best by the user who knows exactly his current needs and the required degree of similarity. However, in principle all of the steps described in the algorithm can be done automatically. If automated support is asked for, the component specification used as a goal description must be a concept (or a realization) described in RESOLVE, to be able to match it to the RESOLVE concepts in the descriptions of components in the repository.

The Component Manager merges both modes, thus providing the user with a hybrid environment. It is up to the user to decide what to do by hand and what to have done by the machine. Therefore, both extreme cases - pure manual and pure automatic retrieval - are still possible.

4.2 Automated Local Matching of Formal Specifications

Automated local matching will allow a user to invoke, whenever necessary or convenient, the automated version of these supporting algorithms providing

```

PROC RETRIEVE-COMPONENT (LIBRARY, COMPONENT-SPEC)
  FOUND := NO-MATCH.
  DETERMINE DECOMPOSITION-LEVEL.
  REPEAT
    DETERMINE USER-VIEW.
    REPEAT
      FIND-SIMILAR (COMPONENT-SPEC, STARTING-POINT)
      REPEAT
        EVALUATE (COMPONENT-SPEC, STARTING-POINT)
        IF (FOUND ≠ MATCH) THEN
          SUBTREE-SEARCH (COMPONENT-SPEC, STARTING-POINT)
        FI.
      UNTIL (FOUND=MATCH) OR
        (NO MORE ALTERNATIVE STARTING-POINTS).
    UNTIL (FOUND=MATCH) OR
      (NO MORE ALTERNATIVE USER-VIEWS).
  UNTIL (FOUND=MATCH) OR
    (NO MORE ALTERNATIVE DECOMPOSITION-LEVELS).
CORP RETRIEVE-COMPONENT.
PROC SUBTREE-SEARCH (CURRENT-SPEC, ANCESTOR).
  WHILE (FOUND ≠ MATCH) AND
    (EXISTS UNEVALUATED DEPENDENT-COMPONENT)
  DO
    EVALUATE (CURRENT-SPEC, DEPENDENT-COMPONENT).
  OD.
  IF (FOUND ≠ MATCH) THEN
    WHILE (FOUND ≠ MATCH) AND
      (EXISTS DEPENDENT-MODULE NOT USED AS NEW-ANCESTOR)
    DO
      FIND-SIMILAR (CURRENT-SPEC, NEW-ANCESTOR).
      SUBTREE-SEARCH (CURRENT-SPEC, NEW-ANCESTOR).
    OD.
  FI.
CORP SUBTREE-SEARCH.

```

Figure 2: Searching in the Component Manager

him/her either with the "second opinion of another expert" or with the help of a "clerk" who is taking over in a well defined situation.

We expect a retrieval style that will utilize especially the automated versions of EVALUATE (decide if a match is given), FIND-SIMILAR (decide on the level of similarity), and SUBTREE-SEARCH (evaluate all components in a given subtree of a user-view's generalization hierarchy); see figure 2. All these processes are, from the point of view of their control structure, straight forward list or backtracking algorithms. The critical point is to decide if and to what degree a match is given.

To decide if a component is a (local) match means to compare the needs statement with the component description. In the case of an automated retrieval, the needs statement is a concept or realization as described for RESOLVE. Thus, the comparison is done on the basis of the formal (RESOLVE) specifications stored in the component manager.

The user of the component manager supplies a specification of the module that is required from the library. The requirement specification does not need to be a complete specification, but the the likelihood of finding the desired component increases with the degree of completeness of the specification. When the requirement specification is matched against a component in the library, the user selects the realization of the component in the library that suits his needs.

Matching one specification against another is in general an unsolvable problem. However, in a great many cases the problem can be solved, at least partially. A grammar for the syntax of concepts has been defined, permitting the compilation of specifications. The assertions are machine processable if the mathematical theories—their notations and operations—are formally defined, and if a base set of logic rules are hard-wired into the processor.

Many specifications describe abstract data type modules. To match two specifications, one can check whether the mathematical models of the two are the same. The number and kinds of parameters to the modules provide other features for comparison. The number of operations can also be compared, as can the interfaces, and the pre-conditions and post-conditions. One approach to making the comparisons is to translate the assertions about the operations' pre-conditions and post-conditions to assertions about a canonical mathematical model, such as set theory. This has the advantage that if a requirements specifier chooses a different model for a data type than the specifier of the library component chose for the same component, matching is simplified. However, translation to a canonical form may increase the number of terms in each assertion, thus increasing the cost of comparison

5 Evaluation of Components - Global Matching

While to match a non-real-time component specification can be done in a local sense, at the interface level, to match a real-time component specification requires more work since the introduction of such a component introduces indirect timing effects on the rest of the real-time application. Predicting real-time performance before the application is actually running is referred to as *schedulability analysis* [14, 16]. Even when done for programs written in languages that conform to time-constrained real-time languages, exact schedulability analysis is NP-complete. Essentially, existing accurate algorithms are exponential in the number of alternate conditional branches found in real-time programs and, in the case of parallel real-time systems, in the number of PEs and software components in consideration. Furthermore, even polynomial-time algorithms may still be computationally-prohibitive, when the number of PEs and components is very large, which is often the case with many modern real-time systems (such as those in the C³I domain of applications, for example).

We make the assumption that the code of components is amenable to static analysis, as in Real-Time Euclid [6]. We assume that loops have been unrolled, that no recursion is used, and that conditionals have been balanced and transformed [17, 18] to eliminate the number of alternate paths schedulability analysis [14, 16] has to consider. We also make the assumption that the call-DAG of components is statically-known. We require that the sizes of all variables and object states be statically determinable. In particular, we employ standard techniques used in RPC and distributed system implementations (such as in SUPRA-RPC [15], among others), to compute the size of each operation parameter. The direction of each parameter (IN, OUT or INOUT) is either available from the language definition or is provided as a remote call annotation. Consequently, each component specification contains sufficient information that describes how much time each operation of the component takes to execute, what other operations of what components each operation of the component calls and how many times, and how much data needs to be transmitted in each direction on each call.

Given the interface description for the component being considered for reuse, a performance estimate of the entire system is undertaken in three steps. First, the demand for each resource (PE, link, sensor and so forth) in the system is projected. For every resource at the node where the component will reside, this demand is computed according to polynomial-time heuristics, which project accurate accumulated execution or communication demand due to every assigned component within a certain interval of time (such as the least-common-multiple of the periods of all real-time processes using the component) and estimate such demand due to the components that have not yet been assigned. (Note that even these are heuristics in the sense that the demands are accurate in the ac-

cumulated sense only and not necessarily at any particular instance of time.) Should there be sufficient time, the same heuristics are applied to nodes and links neighboring this node. In systems with a large number of nodes, the corresponding demands (with the contribution of the component in consideration) are only estimated in the sense that individual nodes are not considered but groups of nodes are, as 'mega-nodes' (one such mega-node includes the node of the component and its immediate neighbors). Should there be a very large number of nodes, then the mega-nodes are combined into "mega-mega-nodes" and so forth.

Once accumulated demands for every resource have been estimated, they are easily converted into utilizations (by dividing over the size of the same time interval over which they have been accumulated). Should any utilization exceed 100%, the component is rejected as too time-consuming for the system. Otherwise, the utilizations are in turn used to compute progress rates incurred by individual or groups of processes when attempting to use a particular resource. Each rate is estimated as an expected value, where the expected probabilities are the probabilities that (1) no request is made for the resource, (2) this process is the only one making a request, and (3) other processes made their requests when this component's request has come. The expected values for each probability are, respectively, 100% (rate of progress), 100% and the fraction of this process's contribution to the total demand for the resource.

Finally, *response times of each process or a group of processes are computed* as sums of ratios of each process's contribution to the total demand for a resource over the rate of progress of the process for this resource, for every resource. The response times are contrasted with the corresponding process periods. Should a response time exceed a period, the performance prediction has identified a potential missed deadline, and the component is rejected as too time-consuming. Should there be multiple components chosen by the same local match, the one which maximizes the laxities (computed as sums of differences between periods and projected response times) in the system is chosen.

To further speed up the performance estimation, demands, utilizations, rates of progress and response times are computed incrementally. Typically, while relatively more work is needed to update the values of these metrics at the PE where the component is to reside, little extra work is needed for the mega- or the mega-mega- etcetera nodes. Furthermore, in systems where reusable component selection is combined with the selection of the node to assign the component to, the performance estimate incorporates a fast assignment algorithm that considers a fixed number of nodes from among the least utilized ones. The overall performance estimating procedure runs in fast polynomial-time, and expected to provide good predictions of run-time real-time performance. A quantitative evaluation of the procedure is in progress.

6 Conclusions

Exploiting the synergy between component library management techniques (the Software Archive [11, 9, 12]) and formal component specification (the RESOLVE specification language [5, 2, 4]), the Component Manager strives to realize a hybrid tool supporting fully automated, partially automated, and manual retrieval of real-time components. Going beyond the level of single components, a possible match is evaluated with regard to its influence on the timing properties of the target system, using the concepts of schedulability analysis.

Ongoing research includes the testing of RESOLVE's practicability for specifications and designs. Work is also in progress to integrate RESOLVE into the prototype of the repository and to refine its real-time extensions to serve the needs of the presented two-stage matching algorithm.

References

- [1] R.J. Brachman. What is-a is and isn't: An analysis of taxonomic links in semantic networks. *IEEE Computer*, 16(10):30-36, Oct 1983.
- [2] W.F. Ogden B.W. Weide and S.H. Zweben. Reusable software components. In M.C. Fovits, editor, *Advances in Computers*. 1991.
- [3] R. Fikes and T. Kehler. The role of frame-based representation in reasoning. *Comm. ACM*, 28(9):904-920, Sept 1985.
- [4] D. Harms. *The Influence of Software Reuse on Programming Language Design*. PhD thesis, The Ohio State University, Columbus, Ohio, Aug 1990.
- [5] D. Harms and B.W. Weide. Types, copying, and swapping: Their influences on the design of reusable software components. *IEEE Transactions on Software Engineering*, 17(5):424-435, May 1991.
- [6] E. Kligerman and A. D. Stoyenko. Real-time euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):940-949, September 1986.
- [7] T.M. Mitchell. Generalization as search. In B.L. Webber and N.J. Wilsson, editors, *Readings in Artificial Intelligence*, pages 517-542. Tioga, Palo Alto, CA, 1981.
- [8] R.T. Mittermeir and W. Rossak. Software-bases and software-archives - alternatives to support software reuse. In *Proceedings of FJCC 87*, pages 21-28, Oct 1987.

- [9] W. Rossak. Software development reusing existing components - the software archive. In *Proceedings of the IEEE PCCC-89, Phoenix Conference on Computers and Communications*, pages 327-331, 1989.
- [10] W. Rossak and R.T. Mittermeir. Structuring software archives for reusability. In *Proceedings of IASTED 5th Internat. Symposium on Applied Informatics*, pages 157-160, 1987.
- [11] W. Rossak and R.T. Mittermeir. A dbms based archive for reusable software components. In *Proceedings of the Second Internat. Workshop on Software Engineering and Its Applications*, pages 501-516, 1989.
- [12] W. Rossak and R.T. Mittermeir. The user-view as a basic concept for software reuse. In *Proceedings of the Eighth International IASTED Symposium on Applied Informatics*, pages 288-291, 1990.
- [13] W. Rossak and L. R. Welch. The component manager: A hybrid reuse-tool supporting interactive and automated retrieval of software components. In *Research Report NJIT/CIS-92-04*, April 1992.
- [14] A.D. Stoyenko. A schedulability analyzer for real-time euclid. In *IEEE Real-Time Systems Symposium*, San Jose, CA, December 1987.
- [15] A.D. Stoyenko. Supra-rpc: Subprogram parameters in remote procedure calls. In *IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, December 1991.
- [16] A.D. Stoyenko, V.C. Hamacher, and R.C. Holt. Analyzing hard-real-time programs for guaranteed schedulability. *IEEE Transactions on Software Engineering*, 17(8), Aug 1991.
- [17] A.D. Stoyenko and T.J. Marlowe. Schedulability, program transformations and real-time programming. In *IEEE/IFAC Real-Time Operating Systems Workshop*, Atlanta, Georgia, May 1991.
- [18] A.D. Stoyenko and T.J. Marlowe. Polynomial-time transformations and schedulability analysis of parallel real-time programs with restricted resource contention. *Real-Time Systems*, 1992. to appear.
- [19] T.J. Biggerstaff and A.J. Perlis. *Software Reusability, Vol. I and II*. ACM Press, 1989.
- [20] L. R. Welch. *Architectural Support for, and Parallel Execution of, Programs Constructed from Reusable Software Components*. PhD thesis, The Ohio State University, December 1990.

The Design of the MARUTI System*

Daniel Mossé, Manas C. Saksena and Ashok K. Agrawala

Institute for Advanced Computer Studies

Department of Computer Science

University of Maryland

College Park, MD 20742

Abstract

The design of an embedded system to control the operations of a large complex real-time system requires a comprehensive framework to meet the diversity of requirements. Such an embedded system must provide support for real-time activities in a fault-tolerant and distributed manner. In this paper, we discuss the design of such a system, called MARUTI, and the guiding philosophy behind it.

MARUTI is a time-driven system, in which resources are reserved for the real-time tasks prior to execution. As much information as possible is gathered about resource and timing requirement of a task so that appropriate temporal resource binding can be done. The resource allocation and scheduling scheme was developed for a distributed system in which each node may be a multiprocessor.

Fault tolerance is achieved through the development of resilient applications in a user-transparent way according to a specified resiliency degree. Active redundancy is used in order to reduce the recovery latency. The resilient applications are allocated onto the distributed system taking into consideration the timing and fault tolerance constraints as well as the characteristics of the distributed environment.

The MARUTI system has been designed to assess the applicability of techniques for real-time, distributed, fault tolerant systems in a cohesive and comprehensive environment.

*This work is supported in part by contract DASG60-87-C-0066 from the U. S. Army Strategic Defense Command to the Department of Computer Science, University of Maryland. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of Defense position, policy, or decision, unless so designated by other official documentation.

Computer facilities were provided in part by NSF grant CCR-8811954.

1 Introduction

Many large, complex systems consist of numerous components which must interact in a controlled manner. An embedded computer system is used to perform the control functions necessary for successful deployment and operation of such systems. In order to meet the control requirements, the embedded computer system must provide support for reliable operation of the software which runs on it. Further, the software must operate within real-time constraints to monitor and generate control signals.

Many of the embedded systems have been designed using the event driven approach in which the system reacts to the events from within or those generated by the environment. Such a system executes the appropriate software in reaction to these events. The events are processed on the basis of priorities assigned statically or dynamically. However, such priority based operation does not always guarantee a timely execution. In a time driven system, all executions are carried out during specific time intervals chosen to assure the timely execution. In this paper we present the basic philosophy behind the design of MARUTI, a time driven system that operates in a distributed environment while assuring the requested fault tolerant behavior.

An embedded system in operation usually executes a limited set of applications in a restricted environment. Not all such systems are closed systems in which all applications and their execution characteristics are known at the design time. Many of them have to accept processing requests made during the operation of the system. The methodology developed to design an embedded real-time system must encompass many characteristics. Since embedded systems have varying requirements, the design must be general enough to be able to adapt to a large variety of operational environments. On the other hand, it must be able to support the implementation of control functions of a specific system in an efficient manner.

When an embedded system has to meet real-time requirements and provide support for fault-tolerant, distributed, and heterogeneous operation, many of the techniques developed for addressing these requirements in isolation are often contradictory. Systems must be designed taking into account all these requirements and integrating their solutions throughout the system design. In MARUTI we are attempting to address these requirements in a comprehensive manner.

The starting point in the design of MARUTI has been a careful consideration of the characteristics of the applications to be supported on it. In the next section we present a brief description of the application characteristics and the resulting system requirements. The design has followed a consistent philosophy which is presented in Section 4. This is followed by the user's view of the application. In Section 6 we present the process to be used in the development and operation of applications taking into consideration all the requirements the applications may have. Some

concluding remarks are presented in Section 7.

2 Related Work

In order to put our work in perspective, in this section we discuss a number of other efforts which are aimed at meeting the requirements of real time operating systems. The discussion is organized along the lines of major requirements and how they are addressed by ARTS[19], Alpha[5], CHAOS[15, 16], HARTOS[9], Spring[18], MARS[7] and RT-Mach[20] systems. Research in ARTS, RT-Mach and Spring is directed towards predictable service in distributed hard real-time environment. CHAOS is designed to support adaptive hard real-time applications, while Alpha is designed for supporting mission critical computing in large, complex, distributed systems with a benefit-accrual model for real-time. Research in HARTOS is focused on fault-tolerant communication in a hexagonal mesh interconnection network.

Hard Real-time: Commercial *real-time operating systems*, such as Real Time Unix (RTU) and LynxOS, address the real-time requirements by providing an interrupt-driven kernel, fast context switching and a priority-driven scheduler. They do not, however, provide hard real-time guarantees.

The Spring kernel and the MARS system are time-driven systems that pre-schedule the critical (hard real-time) tasks. These tasks are the only ones guaranteed to execute within their deadlines. ARTS and RT-Mach use fixed priority scheduling with static schedulability analysis to ensure that deadlines of hard real-time tasks will be satisfied at execution time. CHAOS and Spring support dynamic real-time scheduling for online guarantees.

Distributed and Heterogeneous Operations:

Several of the real-time systems, such as ARTS, RT-Mach, Alpha, Spring and MARS, provide support for distributed operations but not heterogeneity. Spring uses homogeneous multiprocessor hardware with shared memory, while ARTS operates on top of Mach in a distributed homogeneous environment. RT-Mach is a modified version of MACH[1] and also provides a distributed environment.

Fault tolerance: Support for fault tolerance under real-time constraints has not been extensively researched. ARTS provides support for exception handling as well as timing errors whereas Alpha provides data replication as well as process migration for fault tolerance. However they do not address the impact of these techniques on timing characteristics of applications. MARS supports hardware level fault tolerance, using active replication. HARTOS provides for fault-tolerant communication.

Clearly most of these systems address only a subset of the requirements for advanced hard real-time operating systems. The goal of MARUTI is to develop a comprehensive framework for

addressing the hard real-time and fault tolerance requirements in a heterogeneous, distributed computing environment.

3 Application Characteristics and System Requirements

The design of a real-time system takes into consideration the primary characteristics of the applications which are to be supported. Note that these characteristics are derived not just from the real-time applications implemented today but also those anticipated [17]. The characteristics of real-time applications which play a crucial role in the design of the operating system are identified below.

Timing Constraints Real-time applications have various kinds of timing constraints. In hard real time applications, tasks must be completed by the specified deadline for correctness. Overrunning a deadline is considered a failure and cannot be tolerated. Although, soft real time applications also have deadlines, overrun deadlines can be tolerated to a certain extent. A penalty is incurred when a deadline is missed in soft real-time applications. In addition, the real-time system may also be required to execute some jobs which do not have any timing constraints. In many system the hard real-time, soft real-time as well as non real-time applications must coexist.

Criticality Many real time applications are safety critical. Examples of such systems include nuclear power plants, life support systems, etc. A failure to perform the critical tasks successfully can result in disastrous consequences. A real-time system must provide support for fault tolerance and exception handling capabilities for increased reliability and tolerance to failures, while continuing to satisfy the timing requirements.

Distributed Real time applications envisaged today are distributed in nature and some of their components must execute concurrently. A natural way to support this requirement is to have a distributed system, which is a collection of processing nodes connected via an interconnection network. Each node in the system may consist of a variety of resources including one or more processors which may be of heterogeneous architecture. The distribution of the system may also be required to support fault tolerant operation.

Deterministic Execution Profile Hard real time applications require deterministic guarantees, and thus worst case bounds on their execution times and resource requirements must be known for appropriate resource allocation. This forbids the use of unbounded loops and recursion

in the software, which, in general, characterize non-deterministic resource consumption. Further, to prevent unbounded waits, the synchronization of the programs should be known and addressed explicitly.

Scenarios Many real time systems operate in certain well defined modes, which we call scenarios. The system exists in any one scenario at a time but may switch scenarios when a triggering condition becomes true. This requires the system to have the capability to switch between different scenarios.

4 Maruti Principles

Recognizing the complexity of the requirements posed by the characteristics of applications we formulated a few basic guiding principles for the design of MARUTI. In this section we present these principles and a brief rationale for them.

Principle 1 *Time must be an essential attribute of every entity in the system*

In a time driven system, it is necessary that all parts of the system have a uniform and consistent view of time and a way of handling it. In MARUTI, time is treated as an attribute of every entity and all operations are carried out with respect to time. This approach not only assures the time driven operation but also permits reasoning about the temporal properties of the system.

Principle 2 *Do as much work for preparing an application for execution as early as possible.*

It is difficult to predict and account for operating system overheads in a demand scheduling model. In a real time system this adversely affects the required deterministic guarantees. To assure real-time operation the operating system overheads at run time must be predictable and minimized for efficiency purposes. Therefore, during run time, real time operating systems should perform only those tasks that are strictly necessary.

In MARUTI, the application development process goes through several phases. All the components of an application contain the placeholders for all the resource and timing information. The values to these placeholders are assigned as early during the development phase as they can be assessed, refining them as the application development process proceeds.

Principle 3 *All resources needed by a hard real-time application must be reserved prior to execution.*

in the software, which, in general, characterize non-deterministic resource consumption. Further, to prevent unbounded waits, the synchronization of the programs should be known and addressed explicitly.

Scenarios Many real time systems operate in certain well defined modes, which we call *scenarios*. The system exists in any one scenario at a time but may switch scenarios when a triggering condition becomes true. This requires the system to have the capability to switch between different scenarios.

4 Maruti Principles

Recognizing the complexity of the requirements posed by the characteristics of applications we formulated a few basic guiding principles for the design of MARUTI. In this section we present these principles and a brief rationale for them.

Principle 1 *Time must be an essential attribute of every entity in the system.*

In a time driven system, it is necessary that all parts of the system have a uniform and consistent view of time and a way of handling it. In MARUTI, time is treated as an attribute of every entity and all operations are carried out with respect to time. This approach not only assures the time driven operation but also permits reasoning about the temporal properties of the system.

Principle 2 *Do as much work for preparing an application for execution as early as possible.*

It is difficult to predict and account for operating system overheads in a demand scheduling model. In a real time system this adversely affects the required deterministic guarantees. To assure real-time operation the operating system overheads at run time must be predictable and minimized for efficiency purposes. Therefore, during run time, real time operating systems should perform only those tasks that are strictly necessary.

In MARUTI, the application development process goes through several phases. All the components of an application contain the placeholders for all the resource and timing information. The values to these placeholders are assigned as early during the development phase as they can be assessed, refining them as the application development process proceeds.

Principle 3 *All resources needed by a hard real-time application must be reserved prior to execution.*

Clearly, adherence to this principle is essential to guarantee the successful execution of hard real-time applications for which a guarantee can only be given if all resources needed can be made available in a timely manner.

We consider a *semi-dynamic* model for real-time systems, in which a job is submitted for processing prior to its release time (earliest start time). The time between the submission of a job and its release time can be used by the system for carrying out the resource allocation without affecting the timely operation of the application.

In order to accept a real time job for execution, the system performs resource allocation for that job. For hard-real time jobs, if all resources necessary for the execution are available within the time constraints requested, the job is accepted for real-time execution. Otherwise the job request is denied. Soft-real time jobs however, do not need such deterministic guarantees. Non-real time jobs are run on time and resource availability basis and do not require resource reservation. In this paper, we primarily focus on hard real-time operation.

Communication is a vital part of most real-time applications and thus the required resources for communication must be reserved as well. A special case of communication, namely *synchronization* between tasks, is also achieved via appropriate scheduling and resource reservations.

Principle 4 Support for fault tolerance is an integral part of the system.

Most real time applications are of critical nature and operate in an unreliable environment. It becomes imperative that the system provide adequate execution support despite the presence of failures. Fault tolerance is treated as an essential aspect of MARUTI. Support for fault tolerance is uniform, starting at the lowest level and defining error handlers and different plans of actions at each level. The fault handling can be carried out in real-time or non-real time manner as needed by a particular application. These characteristics permit a systematic methodology of execution in a fault tolerant mode, rather than in an ad-hoc manner.

5 User View of Applications

In this section we describe the user view of the applications, i.e., a brief overview of the programming paradigm as well as the capabilities available to the user.

An application program is a collection of cooperating *software modules*. The modules may communicate with each other using shared resources or message passing. Communication through message passing can be synchronous (i.e., remote procedure call) or asynchronous (i.e., one way invocation).

The user specifies timing constraints for the entire program and may also specify timing constraints for individual modules. The constraints for a module can be expressed in terms of absolute time or can be specified relative to the timing constraints of other modules.

The relationship between software modules may be specified as *precedence*, *exclusion* or other constraints such as *simultaneity* (i.e., modules must execute simultaneously) and *indivisibility* (i.e., one module must execute after another with no loss of state from the first module)[22, 14].

Fault tolerance requirements can be specified by indicating the resiliency degree. The resiliency can be specified on an individual module basis or for the entire application program. Upon the detection of a fault, user specified actions (or system defaults) are taken to handle the fault. Mechanisms are also provided to specify exceptions and exception handlers.

A named collection of programs constitutes a *scenario*. Each scenario defines a mode of operation of a real-time system and contains all programs necessary to operate in that mode. Scenarios can also be seen as sudden changes in the executing programs, in response to some stimuli. Scenarios are useful in situations where there are limited resources for execution of multiple programs, or where programs are mutually exclusive.

In our programming paradigm we do not allow nondeterministic behavior, due to the real-time requirements. This imposes some discipline that the programmer must observe (see section 3).

6 Maruti Mechanisms and Structures

MARUTI has been designed using a consistent set of mechanisms and models. In this section we present the framework used in this system. Let us consider the resource model used in MARUTI.

6.1 Resource Model

In our model, the resources are divided into two types: *active* and *passive*. An active resource is capable of autonomous operation, for example CPUs and DMA devices. Passive resources are the storage devices (e.g., memory and secondary storage), which are used by the active resources. Passive resources may be shared and are also used for communication between modules in the same node. Note that both active and passive resources are required for internode communication.

Active resources are partitioned into disjoint groups called MEARGs (Mutually Exclusive Active Resources Groups). Each active resource belongs to exactly one MEARG. The set of active resources available in a distributed system may now be considered as a set of MEARGs. A task¹ executes using the resources of one MEARG and a set of passive resources.

¹A task is a basic executable entity. In section 6.2.2, we show how tasks are obtained from application programs.

The schedule for each MEARG and for each passive resource is kept in a structure called a *calendar*[10]. The calendar can be viewed as a sequence of non-overlapping time intervals, each of which has a task associated with it (the task that will execute during that interval). If a resource can only be used exclusively, an interval can have only one task associated with it.

6.2 Model of Computation

To facilitate resource allocation, the system view of an application program differs from the user view. This distinction comes about due to the different granularity of elements that compose a program. In this section we describe the basic building block of a program, how to create a program, how to add fault tolerance to it, and finally how to allocate resources for it.

6.2.1 Elemental Units

In the system view, the building block of an application program is an *elemental unit* (EU). When an EU is scheduled, we refer to it as a *task*. A task is then the basic entity for execution. A module requires software², state, and resources to execute on a given input to produce output. All these are encapsulated into an EU along with constraints on input and output (Figure 1).

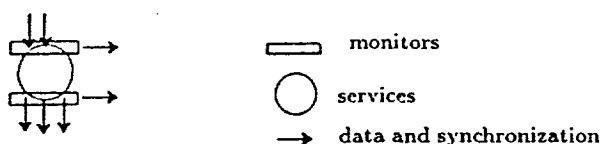


Figure 1: Elemental Unit

Input conditions are boolean expressions on the input data, state, time, and capabilities which are evaluated to trigger services provided by the EU. Similarly, output conditions are checks on outgoing data and state, as well as on the timing requirements of the EU. In addition, input and output data may be correlated before generating results.

The structure of EUs makes it possible to accomplish uniform treatment of both computation and communication services (Figure 1a). For example, a computational EU is typically determined by the data received and generated, by their corresponding validity checks, and by a program that requires a state and hardware resources to operate. A communication EU is analogous, where input and output data are the messages, input conditions may be empty, output conditions check for message corruption, the software is comprised of the data buffering and data movement protocols, and the hardware requirement consists of buffers, communication links (for remote communication),

²Throughout this work we assume that the software is reentrant.

and processor to execute the software. Another aspect of computations, namely *synchronization*, can also be accomplished by requiring the input condition to be satisfied only when all messages have arrived (see Figure 2).

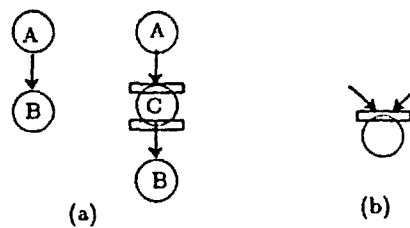


Figure 2: (a) EU communication and (b) EU synchronization

Input and output data are the pieces of information transmitted between EUs or that come from the environment. In addition to the messages, files and other environment data (e.g., current time or other system parameters) can also be considered as input and output data.

Input and output conditions are specified by the user and depend on the data, the state, and the instance of the software module. Temporal constraints can also be checked as part of input and output conditions. The evaluation of conditions can trigger executions, correlate input and output parameters, check validity of states, etc. Note that the evaluation of conditions require hardware resources and the state of the EU. If an error is detected, the appropriate EU is notified, which is depicted by the lateral arrows in Figure 1.

6.2.2 Application Representation

An application program is characterized by a directed acyclic graph called the *Elemental Unit Graph* (EUG), augmented with *timing constraints* and *operational relations*. A vertex in the EUG is an elemental unit and arcs represent control flow.

A task is a unit of execution in our resource model. A task executes using the resources of a MEARG and a set of passive resources. In the system view, tasks communicate using message passing at the end of the task. A module which requires more than one MEARG must be decomposed into sub-modules to conform to the resource model. Optimization techniques like buffering may be used to reduce overheads introduced by such decomposition.

The timing constraint of a task is represented as a 3-tuple, $\langle r, c, d \rangle$, where r is its release time, c its execution time and d its deadline. Time constraints may be defined in terms of absolute times or relative to the time constraints of other tasks. The time constraints of tasks may be derived from the application time constraints.

Operational relations are used to represent synchronization between tasks. Examples of such

relations are *precedence* and *mutual exclusion*. Another relation called *indivisibility* requires the preservation of execution state. For instance in the Figure 3, an indivisibility relation may be defined between A_1 and A_2 , and another between A_2 and A_3 .

Communication between EUs is represented by an EU which requires communication media and other resources for execution. These communication EUs are automatically generated. Heterogeneity is easily supported by performing data translation to a common network representation [3].

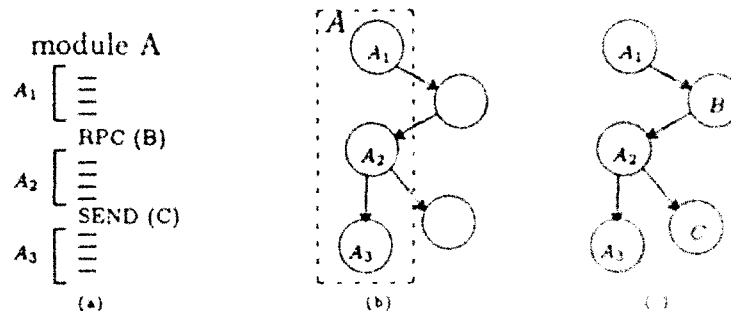


Figure 3: (a) User software modules (b) EU structure (c) EUG

6.3 Application Life-Cycle

The life-cycle of an application program can be divided into *development* and *operational* phases. The application development process goes through several phases before it is ready for operation. In accordance with principle 2, we try to accomplish as much as possible in each phase.

1. **Development Phase:** This phase is broken down into three stages, namely design, compilation, and integration. The result of this phase is an executable application program, represented by an EUG, and which is ready to be submitted for execution.

- **Design.** This stage is the starting point of the development of an application during which the overall design is carried out. The activities during this stage include requirements specification, conceptual design and detailed design.
- **Compilation.** The software modules are created at this stage with the interface specifications. Tools are available at this stage to decompose a software module (Figure 3a) into sub-modules such that each sub-module can be treated as an EU (Figure 3b). The static resource requirements for each EU are extracted at this stage. Figure 3 shows an example of such a transformation. The input and output constraints for each EU may

be generated automatically from the interface specification. The EU structure of the modules is used during the integration stage.

- **Integration.** In the integration stage modules are interconnected to form a program. An EUG is generated from the interconnections specified and the EU structure generated during the compilation stage, as shown in Figure 3c. The remaining resource requirements for the application are identified and recorded with the application, such as the generation of communication tasks between EUs. In order to verify that the interconnections are valid, syntactic type checking on the interface specifications of EUs is carried out at the time of integration.

2. **Operational Phase:** The operational phase consists of resource allocation and execution. For hard-real time applications, we require that resource allocation be done prior to execution.

- **Resource allocation.** When an application program is submitted for execution, the user may specify the time constraints by identifying the *release time* and *deadline*. For periodic applications, the period and termination condition must be provided. An executing program is called a *job*. The allocation and reservation of resources to tasks in a job is performed at this stage. Also, various kinds of synchronization constraints are satisfied intrinsically in this stage [21].
- **Execution.** During this stage the operating system performs dispatching, message passing and reservation enforcement. Previous stages prepare the application for this stage, such that the overheads are minimal. The dispatcher need only examine the calendar and dispatch the tasks whose start time has arrived. In addition, we can guarantee the absence of deadlocks, since there is no waiting.

6.4 Resource Allocation

One main problem we need to address is that of creating the calendars for MEARGs and passive resources so that tasks are executed within their time constraints. While a task requests a single MEARG, there may be several MEARGs capable of meeting this requirement. In general, this leads to a multiple resource management problem which has been shown to be rather complex[6]. Grouping resources in MEARGs reduces the complexity somewhat but does not eliminate it. In a distributed environment the MEARGs may be at different nodes, and each node may contain several MEARGs. To allow efficient allocation of the MEARGs and passive resources, we create calendars in three phases: global allocation, local allocation, and local scheduling.

Global Allocators (GAs) use system-wide resource usage information to decide the node in which a set of tasks is to execute. The *Local Allocator* (LA) carries out the allocation of the resources needed to execute a task, by selecting one among the several eligible MEARGs and some passive resources. The LA is also responsible for coordinating the scheduling of resources to various tasks of an application so that their operational relations can be maintained. The actual interval of execution of a task is determined by the *Local Scheduler* (LS), which makes the necessary entries in the calendar of a resource (MEARG or passive resource). The start and end times of the execution of a task i are denoted by s_i and e_i , respectively³. Also, there is one logical LS for each calendar.

Let us consider the interaction between local allocators and local schedulers. Consider a simple situation where tasks t_1 and t_2 require MEARGs m_1 and m_2 , respectively. Both these tasks are submitted for scheduling in the same multiprocessor node and assume that t_1 sends some data to t_2 through a common shared passive resource R (i.e., the shared buffers must be allocated from the beginning of t_1 to the end of t_2). The LS for R is called LSR. The interaction between the allocators and schedulers is described below.

1. LA sends allocation requests to LSs of m_1 and m_2 for scheduling of t_1 and t_2 .
2. LSs respond with exact start times for execution.
3. LA sends an allocation request to LSR for duration $[s_{t_1}, e_{t_2}]$.
4. LSR responds.
5. On success, LA sends commit to LSs.

Note: If the request is rejected, the LA or the GA may try allocation at some other resource.

A limitation to this approach, however, is that the LSR has no flexibility in the scheduling of the passive resource. Furthermore, the calendars can only be modified by negotiations between the LA, LSR and LSs. This costly negotiation and lack of flexibility, however, can be remedied by management of the passive resources, as follows. In addition to responding with the start times in step 2 above, the LSs also provide the forward and backward slacks associated with each task. The forward and backward slacks of task i (f_i and b_i) are the amount of time a task can be postponed or advanced without violating the timing constraints of tasks in a calendar. The LA sends the scheduled times *plus* the slack times to the LSR. The LSR schedules the passive resource for the interval $[s_{t_1} - \delta_1, e_{t_2} + \delta_2]$, such that $0 \leq \delta_1 \leq b_{t_1}$, $0 \leq \delta_2 \leq f_{t_2}$. The LSR sends δ_1 and δ_2 to LA, which passes it to the LSs along with the commit responses. This scheme allows LS the flexibility to move tasks without the overhead needed to consult with the LSR as subsequent tasks arrive at the LS.

³In this paper we consider only non-preemptive scheduling, such that $e_i = s_i + c_i$.

We have taken the approach of a tiered allocation/scheduling, combined with our resource model, to allow for the scheduling of multiple MEARGs and passive resources for tasks. Two advantages can be identified with this approach. One is that this scheme allows for concurrency in scheduling of individual MEARGs. The LA is capable of initiating multiple local schedulers concurrently without waiting for replies. The second advantage is modularity, which allows testing of different policies at each phase.

6.5 Fault Tolerance

The critical nature of many real-time applications requires resiliency to faults. An important aspect of the MARUTI system is support for fault tolerance. Our main goal is, given a user-specified resiliency degree and a program, to produce a fault tolerant program that will be mapped to the resources during the allocation phase. The resulting job should tolerate faults to the level requested by the user during its execution.

6.5.1 Fault Model

We consider an EU as the unit of failure of an application. In this model the input and output conditions function as the fault detection mechanism. Once a fault has been detected, a fault handling EU may be invoked. Such EU may invoke other EUs to perform recovery and reporting (Figure 4). The fault handling policies in these elemental units can be specified by the user or be a system-defined default, such as aborting the computation and sending a message to the operator's console. For each elemental unit, fault handlers may have different criticalities and execute in different time domains. For example, some may execute in real time (for which resource reservation must be performed) and some others on a resource and time availability basis.

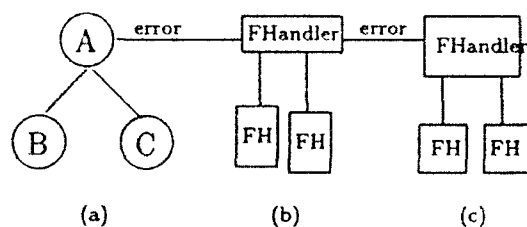


Figure 4: (a) EUG; (b) User fault handlers; (c) Default fault handlers

For fault tolerance purposes, we assume that it is possible to group MEARGs into fault-independent *partitions*, i.e., a fault in one partition does not cause a fault in another partition. This fault independence assumption is valid at a hardware level, taking into account issues such as independent

power sources and different architectures for components. At the software level, independence requires that the replicated versions be of different designs and implementations[2].

6.5.2 Approach

Fault tolerance for real-time systems is not only complex from the point of view of functional correctness, but also due to the timing constraints and resource requirements. In that sense, the scheme adopted to tolerate faults in such systems must take these factors into consideration, as well as the timing requirements of the application as a whole and the user-defined resiliency degree.

Due to the real-time constraints, we use an active modular redundancy approach to construct a resilient application, in a user-transparent way. For each application submitted to the system, the user specifies the required resiliency degree rd . We must ensure that the number of replicas executing the same application is enough to tolerate up to rd faults. The actual number of replicas $rd+l$ depends on the type of faults, the detection scheme, and the system architecture. A discussion on the different values of l is beyond the scope of this paper (see [13, 11] for more details), and in the remainder of our discussion we consider l to be equal to 1.

We do not use roll-back mechanisms such as checkpointing [4], since time is a critical resource that should be accounted for. Roll-back type of methods are applicable to systems where time is not a critical issue [8].

6.5.3 Resilient Elemental Unit Graphs

In Section 6.2.2, we described how to construct elemental unit graphs. In this section we describe how to transform an EUG into a *resilient* EUG (REUG). This process is carried out during the integration stage of the development. The main idea is to replicate the EUs to make the application resilient.

The REUG can be built with *global* or *local* redundancy (Figure 5). Global redundancy is based on replication of the EUG as a whole, where the execution of each replicated EUGs is treated independently. On the other hand, local redundancy replicates specific EUs and can be divided into *total* or *partial* redundancy. Partial redundancy replicates only a subset of the EUs, while total redundancy replicates each and every EU in the EUG. Such flexibility (different types of redundancy) is needed because the EUs in an application might have different probabilities of fault and different criticality.

The software component of each replicated EU need not use the same algorithm design. They may encompass algorithmic alternatives to achieve design independence. Each EU replica is placed in a different partition, to enforce the fault independence at the hardware level. The user-defined

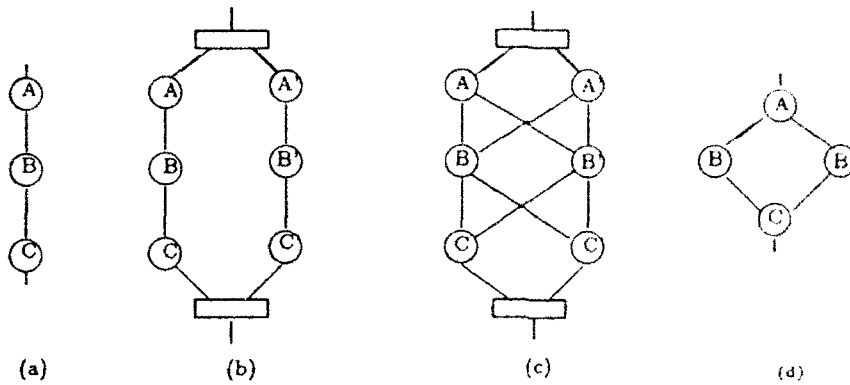


Figure 5: (a) EUG, and REUG by (b) global, (c) total, and (d) partial redundancy

resiliency degree is guaranteed since there is enough redundancy to carry out the application despite the presence of failures.

In order to manage these EU replicas and the communication among them, we associate auxiliary structures, namely *forkers* and *joiners*, with each EU in the REUG. These auxiliary structures are inserted in the REUG automatically and in a user-transparent manner. Consider a message m from task A to task B . The forker for A sends m to all the replicas of B . The joiner of B collects the messages from A and its replicas, and filters them using a message selection algorithm to elect the correct message. It then sends the correct message to the destination B .

A consequence of the use of joiners is that some fault elimination takes place. When the joiner selects one of the incoming messages, and when selection results in the correct message, all incorrect messages get eliminated. Note that the joiners may also recognize the errors in the messages and may be used to trigger corrective actions.

6.5.4 Discussion

The approach introduced above provides fault tolerance in a uniform way, while maintaining flexibility through user-specified resiliency degrees and message selection algorithms. Also, the creation of an REUG and its allocation across partitions of the distributed system is transparent to the user, unless custom fault tolerance is used to create the REUG (partial redundancy). In the partial redundancy case, the user needs only to specify the critical EUs, and the system automatically generates the REUG. In other words, changing the fault tolerance requirements does not affect the functional behavior of the program.

The allocation of REUGs is carried out in a distributed manner [12]. The allocation scheme also enforces that fault detection and recovery are consistent with the real-time constraints, which makes the scheme suitable for real-time applications. In addition, our approach does not require special

hardware. Treatment of faults is an intrinsic characteristic to the application design, due to the EU fault handling structures defined at the application design level. This facilitates maintenance of the system, and allows for changing the levels of criticality of an program (both temporal and functional) after design and integration, but before execution time.

Our model also allows users to achieve a balance between the resource overhead introduced by processing fault-tolerant EUs and the probability of faults. In other words, the user can decide on the cost/reliability trade-off based on the requirements of applications by specifying the resiliency degree for parts of the EUG and the type of redundancy used. Furthermore, It is not difficult to see that the construction of the REUGs are semantic preserving, with respect to the EUGs defined by the users.

7 Concluding Remarks

In this paper we presented the basic structure of MARUTI and the philosophy used in its design. Our experience to date confirms our belief that the comprehensive solutions can only be generated by addressing all the requirements the system must meet at the system design time and developing integrated solutions. For example, in MARUTI the fault handling as well as time handling is carried out uniformly. The fault tolerance capabilities are available in a user transparent way, while permitting the user to enhance the default capabilities if so desired.

The feasibility of the approach taken has been demonstrated through an implementation of MARUTI, at the University of Maryland. The time driven and distributed nature of this design has been established and tested successfully. The fault handling capabilities of the design have been validated for homogeneous architectures.

An interesting aspect of the design of MARUTI is the way it handles the distributed operations in homogeneous as well as heterogeneous environments. In addition, due to its modular design, it lends itself naturally as a testbed for evaluation of the resource management policies. We are in the process of studying various policies applicable to the management of multiple resources, as well as demonstrating the heterogeneous operation of the system.

References

- [1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for unix development. In *Proceedings of Summer Usenix*, July 1986.
- [2] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Trans Software Engineering*, SE-11(12):1491-1501, Dec 1985.

- [3] Andrew P. Balinsky, Ashok K. Agrawala, and Ólafur Gudmundsson. Real-time heterogeneous communication support in maruti. Technical Report UMIACS TR 91-87, CS TR 2697, U of MD, June 1991.
- [4] K. Birman. Replication and Fault-Tolerance in the ISIS System. In *10th ACM Symp on Oper Syst Principles*, pages 63-78, WA, Dec 1985.
- [5] Raymond K. Clark, E. D. Jensen, and Franklin D. Reynolds. An Architectural Overview of the Alpha Real-Time Distributed Kernel. In *USENIX Workshop on MicroKernels and Other Kernel Architectures*, Apr 1992.
- [6] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman Company, San Francisco, 1979.
- [7] H. Kopetz and A. Damm and Ch. Koza and M. Mulazzani and W. Schwabl and Ch. Senft and R. Zainlinger. Distributed Fault-Tolerant Real-Time Systems: The MARS Approach. *IEEE Micro*, 9(1):25-40, Feb. 1989.
- [8] P. Jalote and R.H. Campbell. Atomic-actions for Fault-tolerance using CSP. *Trans Software Engineering*, 1986.
- [9] D. Kandlur, D. Kiskis, and K. Shin. A Real-Time Operating System for HARTS. In Ashok K. Agrawala, Karen D. Gordon, and Phillip Hwang, editors, *Mission Critical Operating Systems*, pages 146-158. IOS Press, 1992.
- [10] Shem-Tov Levi and Ashok K. Agrawala. *Real Time System Design*. McGraw-Hill, New York N.Y, 1990.
- [11] Deron Liang, Yiheng Shi, D. Mossé, and Ashok K. Agrawala. Designing fault-tolerant applications in maruti. In *Proc. 3rd IEEE International Symposium on Software Reliability Engineering*. IEEE, 1992.
- [12] Daniel Mossé, Sam H. Noh, Manas C. Saksena, and Ashok K. Agrawala. Multiple resource allocation and scheduling for multiprocessor real-time distributed systems. In *Workshop on Architecture Support for Real-Time Systems*, San Antonio, Texas, Dec 1991. IEEE.
- [13] Daniel Mossé and Ashok K. Agrawala. Resilient computation graphs for fault tolerance in real-time environments. Technical Report CS-TR-2613, UMIACS-TR-91-29, U of MD, February 1991.

- [14] Vivek Nirkhe, Satish Tripathi, and Ashok Agrawala. Language Support for the Maruti Real-Time System. In *Real-Time Systems Symposium*, December 1990.
- [15] Karsten Schwan and Ahmed Gheith. CHAOS^{arc}: A kernel for Predictable Programs in Dynamic Real-time Systems. In *Seventh Workshop on Real-time Operating Systems and Software*, pages 11-19. IEEE, May 1990.
- [16] Karsten Schwan, Ahmed Gheith, and Hongyi Zhou. From CHAOS^{min} to CHAOS^{arc}: A Family of Real-Time Kernels. In *Proc. IEEE Real-Time Syst. Symp.*, pages 82-92, Dec 1990.
- [17] J. A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Computer*, 21(10):10-19, Oct. 1988.
- [18] J. A. Stankovic and K. Ramamritham. The Spring Kernel. In Ashok K. Agrawala, Karen D. Gordon, and Phillip Hwang, editors, *Mission Critical Operating Systems*, pages 86-117. IOS Press, 1992.
- [19] H. Tokuda and C. W. Mercer. The ARTS Kernel: Toward Predictable Distributed Systems. In Ashok K. Agrawala, Karen D. Gordon, and Phillip Hwang, editors, *Mission Critical Operating Systems*, pages 118-130. IOS Press, 1992.
- [20] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *USENIX Mach Workshop*, pages 73-82, Oct 1990.
- [21] Satish K. Tripathi and Vivek Nirkhe. *Synchronization in Hard Real-Time Systems*. Frontiers in Computing Systems Research - Essays on Emerging Technologies, Architectures and Theories. Plenum Press, 1990. Ed. S. K. Tewksbury, Forthcoming.
- [22] Victor Wolfe, Susan Davidson, and Insup Lee. Supporting Real-Time Concurrency. In *Seventh Workshop on Real-Time Operating Systems*, pages 49-54. IEEE, 1990.

LESSONS LEARNED IN MODELLING FOR ARCHITECTURE ANALYSIS

Jennie A. Strand and Mary S. Tamucci, MYSTECH ASSOCIATES, INC.

and

Dr. Jose L. Munoz, Naval Undersea Warfare Center

ABSTRACT

"Case studies where written up, are rarely presented in forms which reveal.....the full range of difficulties encountered." [1]. The intent of this paper is to document "difficulties encountered" during the past seven years of modelling for analysis of a submarine combat system architecture. Lessons learned, along with resulting recommendations, will be presented. We have divided the modelling process into five phases: Data Acquisition and Archive, Model Design, Model Implementation, Model Exercise, and Results Analysis [2]. The paper is organized to present the lessons learned as they apply to each of these phases. Products and methodologies which are under development or are planned as a result of these lessons will be listed under Products and Methodologies.

INTRODUCTION

The Naval Undersea Warfare Center (NUWC) began using modelling to assess the performance of the architecture of a large, complex submarine combat system at the very earliest stage of the program, that is, virtually while the system was still in the requirements definition phase. MYSTECH ASSOCIATES, INC. (MYSTECH) joined the NUWC architecture modelling team late in 1986 as the competing contractors were preparing their proposals. The modelling team has continued to monitor the development of the system throughout its full life-cycle: from the proposal evaluation, through Full Scale Engineering Development (FSED), and into the test and evaluation phase.

This was an ambitious undertaking considering the magnitude of the new system, and the fact that the architecture team was functionally and geographically separated. Initially, there were those who were hesitant and skeptical about the value of modelling

for architecture performance evaluation. However, as the program developed, and results were achieved, the impact of the modelling on the contractors' proposed designs became evident. Those who had been cautious about accepting the approach began to recognize the contribution of modelling to the system development. Animation of the models provided visual demonstration which facilitated management's comprehension of our effort. The use of operational scenarios as model drivers translated well to operationally oriented program managers.

The utility of modelling was demonstrated to other programs as well. During 1989, NUWC decided to apply the technology to the analysis of a surface ship ASW system. MYSTECH is now supporting NUWC on that program.

The methodologies used to develop the models and evaluate the system architecture are well documented [3,4,5,6]. This paper deals with some of the difficulties encountered over the past seven years, and the lessons learned. The sections of the paper coincide with the five phases into which we have divided the modelling process: Data Acquisition and Archive, Model Design, Model Implementation, Model Exercise, and Results Analysis. Each section presents a brief statement of the lesson (in bold type), a discussion of the difficulties experienced, and a list of recommendations suggested by each lesson. At the end of the paper, methodologies and products which result from the lessons are discussed.

DATA ACQUISITION AND ARCHIVE

A. Data acquisition is the most time consuming portion and has the largest learning curve in the modelling process. "Much time and money can be consumed by what is known as 'getting started.'" [7].

Given that the system under investigation was new system development, and especially during the proposal preparation phase, the majority of the quantitative information and data was unknown. Whatever data was available was typically from "hearsay" and that information was frequently changing. Due to the "fuzziness" of data and the constant fluctuations in levels of information, everything had to be verified and validated before it could be

used. Once the contract was awarded, the data became more substantial but it continued to change frequently. Additionally, most of the information was still at a very high level of abstraction, which is to be expected during the early stages of any system development.

Early on in the program, when quantitative data was not yet available, we learned to use our Best Engineering Judgement (BEJ), as described in [3]. This process enables the judicious application of knowledge gained on previous, or like, systems developments to the current system under evaluation. All the data developed using BEJ needed to be verified. The prime directive was: "Do not use data you do not understand" [8]. The search for information to verify these assumptions was very time consuming since there was no central repository.

General combat systems knowledge, as well as specific data concerning elements within the proposed competing architectures, was gathered from as many areas as possible. Sources included engineers and documentation from other related NUWC programs. Acoustic, database management, and simulation experts provided a rich source of information. Product literature on operating systems and database management systems, specifically those under consideration in the proposals, provided baseline data for the simulations. As much information as possible was gathered from whatever documentation was received from the competing contractors, and subsequently from the prime development contractor after the award.

Recommendations:

1. Use of "best engineering judgement" to fill in the holes is a useful and productive process when applied with extensive record keeping and verification. This allows the project to move forward while the painstaking process of data gathering is going on.

2. An extensive system-specific document library was developed which contained all documentation received during competition and then during FSED. As each document or memorandum was received, it was catalogued and stored, and a notice was distributed to all team members.

3. The ability to reuse documents/knowledge/data from one program to the next saves valuable time and effort. When talking about the reuse of problem solving information, Rumbut emphasizes the value of "reusing specification data from previous development efforts" [9]. Knowledge gained from the submarine combat system architecture evaluation provided valuable support, and shortened the start-up time, when the modelling process was initiated on the surface ship ASW system. Specifically, knowledge of the communication network, methodology for designing the model, and the actual skeleton of the system model were all re-used.

4. Data acquisition must be started as early in the modelling process as possible. As much documentation as possible was gathered as quickly as possible to begin development of a surface ship ASW system document library.

B. Much time can be consumed waiting for all the data for the development of a complete and accurate model.

In preparation for evaluation of the proposed submarine combat system designs, several NUWC engineers were tasked to define a notional architecture to model nearly ten months before the time that the NUWC/MYSTECH team was assembled. As mentioned, there was very little data available, especially at that point in the program. The team finally did build a skeleton of a system model, nearly a full year after the original tasking. Nine months later, a memo documenting model results was written containing the statement "Results are considered preliminary because of the many assumptions and 'best engineering judgements' incorporated in the model" [10]. Two years into the project, 50% of the messages in the Interface Requirements Specification (IRS) were still incomplete.

At times, the system model contained varying levels of detail as system component data became available at different times. For instance, there was detailed information about the communication network protocols, but very little was known about the workstations and data manager. Jenevein notes: "The model must deal well with different levels of design abstraction components which are not to be immediately implemented can be left in abstract form and interfaced to the more resolved component models" [11].

Recommendation:

1. Models were developed in a top down fashion, starting with simple, high level models and using BEJ where needed. All assumptions and engineering judgements were documented and included with model results. As data became available, corrections and amplifications were made and the models became more detailed. In this manner, we were able to evaluate the proposed system architectures with a degree of confidence, especially during the proposal preparation phase, despite the lack of information.

C. Careful records of assumptions and data must be maintained for validation and verification (V&V). Without traceability, chaos can result.

The assumptions and best engineering judgements used to develop our models were replaced with actual data as it became available. As the system design evolved, data such as system topologies, bus rates, message rates and lengths, maximum message and packet lengths, and communications network queue priorities, changed frequently. Team leaders cautioned: "Don't throw anything away - things are constantly changing" [8].

Two models were developed simultaneously, one for each of the two competing architecture designs. Thus, it became imperative that information for the two models, sometimes very similar, be kept separate and distinct. Our goal was to evaluate the architecture of the proposed systems. Model results were one of the means to achieving this goal. For results to be credible, we had to be able to justify the data used to produce them.

The models shifted rapidly from proposal evaluation models to FSED models. Some of the models were over four years old by that time. We needed to be able to retrieve and justify any data which was used in the original models or model designs. By the time FSED models were being developed, enough information had been archived, and enough models had been built, to enable a structured V&V process.

Our model results were compared with the results of the development contractor's own modelling efforts, so all assumptions, engineering judgements, and data used to develop the models had to be documented for identification and verification. Comparison of

results from the two parallel efforts provided additional V&V, and encouraged dialogue between the two organizations.

A database was maintained containing thousands of messages used to drive the models. We created message groupings by combining messages by source and destination, and then by rate, and executed the models with these groupings. Despite the careful recording of which messages were used in which model and for which run, there was a problem tracing the origination of some messages and their rates and sizes. We actually had an internal lessons learned meeting at the time to review what had gone wrong. The finding was that there still was not enough record keeping and data archival.

Recommendations:

1. The importance of a catalogued document library was emphasized. This facilitated the retrieval of data sources, and provided traceability.

2. An extensive database of the messages, operational events from the scenario, system software modules, and processor characteristics was developed and maintained.

3. An engineering notebook was developed and maintained for each model. Meeting notes, memos, model designs, assumptions, BEJs, questions, issues, results of all runs, and analysis of the results were recorded.

4. Naming conventions were established for modules, messages, and hardware devices for use within the models.

5. A Standard Template for Software Development specified preamble standards for documentation of each module [12]. Items included in this template were:

- a. routine name and purpose
- b. author
- c. release date
- d. modification dates
- e. inputs and outputs
- f. global and local program variables
- g. subroutines that called this routine

- h. subroutines called by this routine
- i. change list, including date and description
- j. special testing considerations.

6. Several standards were developed to "ensure that all models produced.....form a consistent set of tools" [13]. These standards included module execution time estimates [14], message rate groupings, channel characterizations, and network delay and overhead rate calculations.

7. The models were designed to make extensive use of input files, which made the data easy to identify and retrieve. Input files were developed for software processes, hardware devices, messages, system topologies, and scenarios.

MODEL DESIGN

A. First, know and understand the model goals. That is, what information are you trying to get from the model?

The architecture lead engineer admonished us to "wear a system engineering hat" and to remember what our requirements were, that the "goal was to evaluate the architecture, not develop a model" [8]. This was especially important because we were evaluating a system throughout its development process, and the overall program issues changed as the program matured.

We got immersed in a modelling problem in which the number of modules (M) increased tremendously based on the number of nodes (N) in the topology ($M=N(3+2N)$). Considerable manhours were spent looking for possible solutions. Finally, we asked ourselves "what are we trying to accomplish in this modelling effort? what questions do we want to answer?" [8].

Recommendations:

1. A set of metrics for system architecture performance evaluation were defined. These came to be known as "the big eight" and were incorporated into the goals for every model:

- a. resource utilization
- b. system throughput

- c. message response time
- d. data latency
- e. data senescence
- f. system expandability
- g. system reconfigurability
- h. identification of system 'choke points'.

These measures were useful in evaluating system behavior under various reconfiguration conditions, and to study failure modes and possible recovery actions. As the evolution of the system brought about different critical system issues, these eight metrics remained the overall modelling objectives.

2. To prevent the models from "taking on a life of their own", thus becoming unresponsive to program objectives, a structured review cycle was developed to be followed for each of the models. This consisted of several structured steps which all members of the modelling team were expected to follow:

- a. initial strategy meeting
- b. requirements/goals review
- c. high level design review
- d. detailed design reviews
- e. model implementation
- f. results review
- g. conclusions and recommendations report.

Throughout this cycle, the models' goals were kept visible to ensure that the models were attaining them.

B. A model must be tailored to a specific area of interest: too broad is as undesirable as too much detail.

The initial plan for the system model was to include every element of the system architecture. This would have produced a model which was too big, too complex, and virtually unmanageable. "We had a concern that our model was too complex but that a simpler model would reduce the sensitivity." [8].

The modelling team was directed to develop a smaller, detailed model for each individual subsystem within the combat system; at the same time still develop a larger system model. It was not immediately clear how to separate the detailed models from the

system model. Even though a system topology diagram was used to determine where the dividing lines should be, there was still much confusion. It was difficult to determine how to partition the system elements. It also became unclear as to what the "system model" actually was [15].

Recommendations:

1. Looking at the system from both directions at once (i.e., from the top down and from the bottom up) gave valuable insight into the important details. This produced a reliable set of partitions. For example:

a) Implementation of the partitioning guidelines defined in [3] provided some structure which eliminated the confusion. Using a top down approach, a high level system model was defined with black boxes for the subsystems. In addition, a Register Insertion Ring Network (RIRN) model was developed using black boxes for each node. Later, a detailed node model was developed consisting of three segments, one for each processor within the node.

b) Reversing the process, a bottom up approach was also used. For each competing contractor's architecture, detailed models were developed of the data manager, system network, system executive (operating system), signal processing, and workstation. These detailed models were then abstracted and represented as black boxes in the system model.

2. Definition of interfaces between models is critical. Clearly defined interfaces between separate models must be ensured if they are to be connected to a system model at some point. One important guideline that was imposed was "not to hook things that are not really elements of the real system so they don't corrupt the data flow" [8]. Statistical distributions were used to represent each detailed model and that element's impact on the system. Use of a common set of scenarios to drive all the models provides a consistent foundation between the detailed models and the system model.

C. Must ensure that the model is easy to modify and maintain.

The assumptions used in the models changed constantly, therefore had to be easily visible and identifiable. During the very early stages of the program, not only did topologies change frequently, but the actual hardware elements of the system changed as proposed designs evolved. In fact, the specific database management system and operating system were not chosen until well into the FSED phase of the program.

Since the models were used to perform parametric studies, parameters to be varied had to be easily accessed. During proposal evaluation, rapid prototyping was performed to support NUWC's requests for certain types of simulation runs where results were needed within 24 hours.

During that time, the simulation language of choice was CACI's NETWORK II.5® [16], which was very easy to use. However, one of its limitations was that every hardware device in the model had to have its own separate hardware device definition and software module. Since the system model had dozens of nodes which were all physically and functionally identical, a single hardware definition/software module pair should have been sufficient to describe all the nodes. Instead we had to define a separate pair for each node, with the only difference being their names. Every time a node changed, which happened frequently, numerous identical modifications had to be made. Typographical errors were prevalent due to the similarities within the names.

Recommendations:

1. The simulation language of choice changed from NETWORK II.5 to CACI's SIMSCRIPT II.5® [17], which provided for greater model complexity while allowing ease of modification; both of which negated the cost of greater model development effort.

®NETWORK II.5 is a registered trademark and service mark of CACI, Inc. - Federal.

®SIMSCRIPT II.5 is a registered trademark and service mark of CACI, Inc. - Federal.

2. Extensive use of input files was made to contain much of the frequently changing data. There were files to represent several different model parameters:

- a. message files contained sources, destinations, rates and lengths
- b. hardware files contained processor speeds and bus speeds
- c. software files contained processing times and SLOC sizes
- d. operational scenario files contained operator actions and tactical event descriptions
- e. topology files contained architecture topology descriptions
- f. application to device files contained detailed software to hardware mappings.

D. Model design must provide for use of parametric loading as well as realistic operational scenarios.

The system model was used to perform parametric studies with varying message rates, lengths and iteration periods. These studies were used to measure response times over different network paths. Messages were generated periodically, either with some statistical distribution or actual system intervals. We called these simulations "time driven". In some of these studies, the model was also loaded with background "noise" in which statistical distributions were used to simulate network traffic and device processing delays. Specific test messages were then injected on top of this "noise" to measure critical paths and actual message delays.

Realistic operational scenarios were constructed to provide insight into the behavior of the system architecture under different levels of operational stress. The system model was driven by several stress scripts in which actual system messages were transferred over the network and software processing was incurred as realistically as possible. The events simulated included aperiodic operator actions such as console button pushes, which incur a specific system response. Most of the operator actions also caused a chain of events to occur in which messages were generated asynchronously. We called these simulations "event driven". In addition to the scenario events, some of these

simulations also modelled the overhead incurred by such background processing as database management, PM/FL, the operating system and resource management.

Recommendations:

1. Construction of input files to define the varying parameters for the parametric studies is useful. In this way the code will not require modification for each study.

2. The messages to be used for time driven simulations should be contained in input files. Scenario inputs for event driven simulations should be contained in input files as well. These should include not only the initiating operator actions, but also the resulting chains of events (i.e. processing times and messages).

3. Use operational scenarios to exercise the models. This provides not only a common source of data to be used by all the models and model developers but further reinforces the model's ability to accurately evaluate the system for its intended purpose. In addition, because these scenarios are developed using combat system terminology, models can be more readily explained to other system engineers involved in the program and more readily appreciated by management. It is easier to describe the architecture's behavior in the context of something such as system IPL rather than "getting a message from A to B via nodes X, Y, Z".

4. Models should be designed to support event driven simulations and their asynchronous operational events and messages. This provides insight into the behavior of the system under various realistic stress conditions.

MODEL IMPLEMENTATION

A. May need to use different modelling techniques and tools depending upon the elements to be modelled.

Early in the program, before either competitor had settled on an acoustic front end architecture, NUWC chose to model a representative cabinet associated with a deployed sonar system to explore the dynamics of that data processing. This was a good

first model since there was existing data and results could be reasonably well predicted. The model results agreed with spreadsheet calculations, confirming belief that this processing is pipelined and did not require dynamic modelling.

One model of the entire combat system architecture would have been too large if all the details were included. Even with careful application of methods defined in [3], the system model became increasingly large and complex for NETWORK II.5. It was difficult to modify and replicate modules, and simulations took too long.

Recommendations:

1. Tool selection should be tailored to the characteristics of the elements to be modelled and the model goals. Dynamic models are often not necessary, nor are they always productive, when a static model of the element can be used. Indeed, early in the process spreadsheet type models might be preferable.

2. Statistical distributions can be calculated, using a package such as UNIFIT® [18], to represent the detailed models of system components in the system model.

3. Be prepared to change simulation languages as models mature, or goals change. For example, SIMSCRIPT II.5 replaced NETWORK II.5 as the models became increasingly complex.

B. Be able to support portability of models across various platforms.

The NUWC/MYSTech modelling team was geographically dispersed at various sites in Rhode Island and Connecticut. Models were developed and run at each of these sites, but on different machines.

Some of the simulations, being run on a VAX 11/780 and VAX 11/785, took several days to complete. In an effort to reduce execution time and increase performance, the models were ported to SUN workstations which produced faster run times. Eventually, two of the models were ported to a CRAY where run times were reduced to hours instead of days.

*UNIFIT is a registered trademark of Select Software Services.

Animation was used as a debugging aid, to enhance our understanding of how the architecture behaved and to support demonstrations. To meet that need, the models were ported to Sun workstations

Recommendations:

1. The benefits of portability became apparent very quickly as models were developed on different machines at different sites. The ability to use the SUNs and a CRAY to shorten simulation run times was crucial for providing rapid turnaround of model results. The use of SUNs, and PCs if possible, to support animation provides a valuable way to instruct model users and program managers who are otherwise unfamiliar with the models or modelling tool. Furthermore, no programming changes should be required to move from one platform to another. Otherwise, there will be additional configuration management impacts. The simulation language must be chosen carefully at the outset to ensure ease of portability later.

MODEL EXERCISE

A. Must have good configuration management in order to replicate runs.

We performed numerous runs with each model due to erroneous or changing data. The models were used to perform parametric studies and rapid prototyping. They were executed under a variety of different operational scenarios. Despite the engineering notebooks, naming conventions, input files and modelling standards, it was still a difficult process to keep track of each run: its purpose, unique parameters, run time and duration.

Recommendations:

1. For parametric studies, SIMSCRIPT II.5's reset statement was used to perform multiple runs with one batch job. This statement reinitializes all statistical counters relative to the listed variables. The entire parametric study could be performed with one simulation by resetting the selected parameter and all the statistics for each run. Selected simulation tools should have this capability.

2. A directory tree structure was developed for organizing each model, its runs, and its unique input files. All common input files were stored in one directory. Full path names were specified within the models.

3. Output files were designed such that all initialization parameters for a given simulation, including all input file names, were echoed to the output files at the beginning of the simulation. This design was incorporated into the surface ship ASW system models as well.

RESULTS ANALYSIS

A. Thorough V&V of the model and its results must occur throughout the life cycle of the model.

As would be expected, a thorough analysis of simulation results occurred after the initial runs to V&V the system model. However, as the model was undergoing numerous, however minor, modifications and being used for rapid prototype runs, not all the modifications were thoroughly verified for their impact on the model. One set of results showed high utilizations on certain processors but upon further research, we determined that there were errors in some message iteration periods and other messages were being transferred from the wrong source. In another instance, large queues ended up being the result of typographical errors in source and destination names. Once we animated the model, we re there was an error in the way that the message protocol sequ was implemented.

Recommendations:

1. The structured review cycle described under Model Design can be applied. Although it may be impractical to apply to minor model modifications, ensure that all major design changes and, of course, any new models follow the review cycle.

2. Maintain engineering notebooks and ensure all model designs, design changes and model load files are recorded in the notebooks.

3. Divide the modelling team into three groups: scenario developers, model developers and model analysts. The scenario developers design all scenarios to be used to drive the models and define the messages and processing events which should be included in each operational event sequence. The model developers design and develop all the models. The model analysts analyze all the simulation results for model V&V, and for performance evaluation of the system being modelled. In this way, the scenario developers become responsible for the accuracy of the messages and processing events which are to be loaded onto the model. The model developers become responsible for implementing an accurate system simulation. The analysts become responsible for ensuring the accuracy of the results and determining the causes of any variations or anomalies.

B. Steady state and transients may cause problems in results if not well understood.

The model runs had to be of sufficient duration to ensure that steady state was reached. Steady state problems occurred when the system model was loaded such that all periodic messages were transmitted starting at time zero. This caused a heavy initial load on the system, so we had to wait until the effects of this had settled out and the messages began their individual periodic generation cycles.

The analysis of particular model runs revealed large message queues at some nodes. It seemed that the model had uncovered a system bottleneck. Further investigation determined that the cause of the large queues was several long messages being transferred from those nodes. Upon additional analysis, it was discovered that these messages did not travel over the network and should not be included in system network evaluations. A thorough investigation of the transient large queues deterred us from perhaps erroneously reporting the existence of a system bottleneck.

Recommendations:

1. Use of periodic snapshots during the model runs supports assessment of when steady state has been reached.

2. The modelling language's reset option should be used to reset the statistics during the simulations after the initial heavy load of messages has occurred.

3. To alleviate some of the initial heavy loading problem, develop start time rules for when to start periodic messages.

4. Dividing the modelling team into subgroups as described under Results Analysis, facilitates the ability to uncover problems observed in the results. The separation of the team into scenario developers, model developers and results analysts ensures distinct allocation of responsibilities. This supports the "retracing of steps", from statistical output to model input or model design, needed to understand transient results.

C. Must use caution when comparing results of different models.

When the project began, there were three separate modelling efforts being conducted in parallel: each competing contractor developed models of their own proposed architecture; NUWC also developed a separate model for each competing architecture. This meant four different models. The modelling languages used for each were not necessarily the same. The problem became: how do we compare the models' results and make sure that the comparisons are valid and fair?

An operational scenario consistent with doctrine and the intended use of the system was developed. A set of stress scripts which depicted system load at various times during a mission were extracted from the scenario. These scripts provided a common baseline for loading and running all of the models. This enabled the modelling team to evaluate results from different models of the same architectures using the same set of metrics.

After source selection, two parallel architecture modelling efforts were conducted by NUWC and the development contractor. Results were compared. In some instances, the results conflicted but it turned out to be caused by either the use of different assumptions or a difference in model implementation. A conflict in resource utilizations was found to be caused by the addition of messages which did not in fact travel over the network, and the implementation of multicasting communications within the NUWC model. Higher transfer delay times were due to an erroneous implementation of a data packet transfer through a forwarding node. Once changes were made, the results were consistent between the models.

Recommendations:

1. The initial choice would seem to be to use the same modelling language. However, the goal of developing independent models of the architectures is to verify that the contractor is in fact designing the system as proposed and required. This can be verified via verification and validation of the contractor's models. If the assumptions, data, designs, and implementations are accurate, then results from separate models, even in different languages, should be consistent. A danger of using the same language is that one of the modelling teams can easily end up duplicating the other team's models without even being aware this is happening. Hence an independent evaluation of the system does not occur.

2. Using a common set of realistic operational scenarios to drive the models provides a common baseline for comparison between the government's predictions and the prime development contractor's. This forms the basis for a common understanding of what is expected, a consistent set of results, and a forum in which to discuss resulting system issues, rather than dwelling on the "ones and zeros".

GENERAL LESSONS

There are additional lessons, which did not result from difficulties encountered, but became emphasized over the course of the program. The NUWC/MYSTech team consisted of people with varying levels of experience and education. There were project managers, systems and software engineers, operational analysts, and junior programmers. Years of experience ranged from over twenty years to new hires fresh out of college. There were experienced modelers, and some with no experience at all. Despite this diversity, the effort of designing and developing the models taught two very valuable lessons:

A. Even if the models had not been built, the amount of information gathered on the new system design, and the insight gained from dissecting the architecture into its various components for close scrutiny provided valuable support to NUWC's Technical Direction Agent responsibilities. Model development is a good

educational tool to support learning about the intricacies of a system quickly.

B. The assignment of different tasks to different people focussed their efforts. The definition of model goals provided a direction to follow while uncovering the necessary data. The interfaces between the detailed models and the system model, and the structured review cycle, ensured the flow of information among the team members so that everyone shared the same general level of system understanding. The value of the TEAM was emphasized. The sharing of information across functional and organizational boundaries allowed NUWC to examine system interfaces and identify critical areas of risk which may not otherwise have been discovered. Important dialogues were initiated which will continue for the next generation of systems development.

PRODUCTS AND METHODOLOGIES

Examination of the lessons learned over the past several years indicates several voids in modelling and analysis activities which we are now beginning to fill. Several products and methodologies are being developed which capitalize on the lessons [19]. This section describes those products and methodologies.

- Develop an information base to store all the data as it is acquired. Organize the data so that it can easily be accessed for other projects and models.
- Add to the information base all system performance requirements and model data to facilitate configuration management.
- Develop a library of reusable simulation objects that can be expanded as new models are developed. Include within the library analytical and simulation tools and languages, and in particular, include an object-oriented programming language.
- Add to the library any simulation and analytical models which can be reused on different modelling tasks.

- Develop a Concept Assessment Tool to aid the modeler in organizing and learning top level system requirements.
- Design the Concept Assessment Tool such that it becomes an automated method for composing a set of measures of effectiveness from system performance requirements. Tying evaluation criteria to specific system requirements allows validation of model goals.
- Develop a run time control facility to assist in setting up model runs and submitting and controlling the runs. The facility would also support the creation of a run library containing all the elements required to replicate a run.
- Utilize the run time control facility to assist the modeler in preparing the runs, and to provide configuration management of the model runs.
- Develop an analysis/V&V facility to assist the analyst in evaluating model results. Utilize animation capabilities to assist in model V&V.

SUMMARY

Difficulties encountered over the past seven years of submarine combat system architecture modelling, and the lessons learned along with their recommended solutions, have been explored. Several products and methodologies for avoiding the same difficulties (or at least being better prepared to meet them) in the future have been discussed. Future plans for application of these products and methods are being developed. These plans include:

- Development of an integrated simulation environment which would include facilities for scenario generation, model development, and results analysis. The simulation environment should provide all the tools needed for accurate and thorough assessment of an architecture through simulation.

- Development of a graphical interface to the simulation environment which would include a model user facility to allow program managers and system designers to perform analysis and evaluation without needing to know the details of the models themselves.

REFERENCES

- [1] Balmar, David W., and Paul, Ray J., "Integrated Support Environments for Simulation Modelling", Proceedings of the 1990 Winter Simulation Conference.
- [2] MYSTECH Internal Memorandum, "System Architecture Engineering Methodology", 1988.
- [3] Munoz, Jose L., "Divide-and-Conquer: Modelling for Architecture Assessment", CACI Simulation Conference 11, September 1987.
- [4] Munoz, Jose L., "Modelling for Architecture Assessment", Tools for the Simulation Profession, 1988.
- [5] Harrison, Steve, "Modelling and Analysis with Databases, Scenarios and Simulations (MADSS)", CACI Simulation Conference 12, August 1988.
- [6] Law, Averill M. Dr., and Kelton. David W., Simulation Modelling and Analysis, McGraw Hill Book Co., 1982.
- [7] Rine, David C., "Improving Software Productivity: An Expert Database Approach", IEEE Expert, Summer 1988.
- [8] Strand, Jennie A., Personal Notes, AN/BSY-2 Project, 1986-1989.
- [9] Rumbut, John T. Jr., and Rumbut, Cindy W., "New Approaches for Large Real-Time Embedded System Specification", Proceedings of the Systems Evaluation and Assessment Technology Workshop, August 20-22, 1991.

REFERENCES (continued)

- [10] Krzych, M., "Preliminary Results of Architecture Processing Modelling and Analysis for AN/BSY-(2) Proposed Architecture", NUWC Memorandum 72153/295, December 11, 1987.
- [11] Jenevein, Roy; Turpin, Russell; Neuse, Doug; Weller, Sawyer; Rao, Mohan; and Browne, J. C., "Critical Component Analysis and Synthesis for Hardware/Software Systems", Proceedings of the Systems Evaluation and Assessment Technology Workshop, August 20-22, 1991.
- [12] Kuznitz, M., "Standard Template for Software Development", NUWC Memorandum 82153/167, June 13, 1988.
- [13] Ionata, J., "Representation of Module Execution Times in Detailed Models", NUWC Memorandum 72153/78, March 26, 1987.
- [14] Ionata, J., "MC68020 Standard Instruction Size", NUWC Memorandum 72153/82, March 30, 1987.
- [15] Munoz, Jose L., "Modelling and Models", NUWC Memorandum 72153/159, June 2, 1987.
- [16] CACI, Inc., NETWORK II.5 User's Manual Version 3, CACI, Inc., La Jolla, CA, 1987.
- [17] Russell, Edward C., Building Simulation Models with SIMSCRIPT II.5, CACI, Inc., Los Angeles, CA, 1983.
- [18] Law, Averill M. Dr., and Vincent, Stephen G., UNIFIT Users Guide, Simulation Modelling and Analysis Co., 1985.
- [19] Tamucci, Mary, "Shipboard Communications Systems Network Architecture Modelling CDRL A002", MYSTECH Document No. D-22-91, June 28, 1991.

DISTRIBUTION

	<u>Copies</u>		<u>Copies</u>
DEFENSE TECHNICAL INFORMATION CENTER	12	ALLIED-SIGNAL AEROSPACE ATTN MICHELLE C HUGUE	1
12 CAMERON STATION		DAR-TZEN PENG	1
ALEXANDRIA VA 22304-6145		JEFFERY ZHOU	1
LIBRARY OF CONGRESS		9140 OLD ANNAPOLIS RD	
ATTN GIFT AND EXCHANGE DIVISION	4	COLUMBIA MD 21045-1998	
WASHINGTON DC 20540		ASCENT LOGIC CORPORATION	
OFFICE OF NAVAL TECHNOLOGY		ATTN JOHN LONG	1
ATTN CODE 227		180 ROSE ORCHARD WAY #200	
(ELIZABETH WALD)	1	SAN JOSE CA 95134	
(GRACIE THOMPSON)	1	AUTOMATED SCIENCES GROUP INC	
800 N QUINCY ST		ATTN RAJIV JAIN	1
ARLINGTON VA 22217-5000		CHARLES ROBERTSON	1
CENTER FOR NAVAL ANALYSES	2	PO BOX 1750	
4401 FORD AVENUE		DAHLGREN VA 22448	
PO BOX 16268		CHARLES DRAPER LABORATORY INC	
ALEXANDRIA VA 22302-0268		ATTN MS 4E (D ALLINGER)	1
ADVANCED SYSTEM TECHNOLOGIES		555 TECHNOLOGY SQ	
ATTN ROBERT T GOETTGE	1	CAMBRIDGE MA 02139	
GARY WRIGHT	1	COMPUTER COMMAND AND CONTROL COMPANY	
5113 LEESBURG PIKE 514		ATTN EVAN LOCK	1
FALLS CHURCH VA 22041		NOAH PRYWES	1
AEPCO		2300 CHESTNUT STREET SUITE 230	
ATTN JOHN BURCH	1	PHILADELPHIA PA 19103	
WASSIL SKILSKYJ	1	CONCEPTUAL SOFTWARE SYSTEMS	
15800 CRABBS BRANCH WAY		ATTN EDP ANDERT	1
SUITE 300		P O BOX 727	
ROCKVILLE MD 20855		YORBA LINDA CA 92686	
ALLIANT TECHSYSTEMS		CORNELL UNIVERSITY	
ATTN TONY NIOLU	1	ATTN KEITH MARZULLO	1
6500 HARBOUR HEIGHTS PARKWAY		UPSON HALL	
MUKILTEO WA 98275		ITHACA NY 14853	

DISTRIBUTION (CONT.)

	<u>Copies</u>		<u>Copies</u>
DUKE UNIVERSITY ATTN KISHOR S TRIVEDI DURHAM NC 27706	1	KAMAN SCIENCES CORPORATION ATTN ROBERT L VIENNEAU 258 GENESEE ST SUITE 103 UTICA NY 13502-4627	1
EG&G-DC ASC ATTN THOMAS A ESCALANTE 1900 DAHLGREN RD PO BOX 552 DAHLGREN VA 22448	1	MYSTECH ASSOCIATES INC ATTN JENNIE A STRAND PO BOX 220 MYSTIC CT 06355	1
GOVERNORS STATE UNIVERSITY ATTN SUNG YOUNG LEE COLLEGE OF ARTS AND SCIENCES UNIVERSITY PARK IL 60466	1	MYSTECH ASSOCIATES INC ATTN JOSEPH A TAMUCCI MARY S TAMUCCI 5205 LEESBURG PIKE SUITE 1200 FALLS CHURCH VA 22041	1 1
GE CR&D ATTN DAVID OLIVER P O BOX 8 SCHENECTADY NY 12301	1	NAWC ATTN JESSE WILLIAMS WARMINSTER PA 18974	1
GRUMMAN CORPORATION ATTN JOHN LITKE STEPHANIE WHITE BETHPAGE NY 11714	1 1	US NAVY ATTN CLIFFORD A WHITCOMB 18 TADMUCK RD CHELMSFORD MA 01824	1
HONEYWELL SRC ATTN MN65-2100 (P BINNS) 3660 TECHNOLOGY DRIVE MINNEAPOLIS MN 55418	1	NAVAL AIR DEVELOPMENT CENTER ATTN TIM MONAGHAN WARMINSTER PA 18974	1
IBM T J WATSON RESEARCH CENTER ATTN FARNAM JAHANIAN PO BOX 704 YORKTOWN HEIGHTS NY 10598	1	NAVAL POSTGRADUATE SCHOOL ATTN CODE EC/LE (C LEE) CODE AS/RA (B RAMESH) CODE AS/SS (N SCHNEIDEWIND) CODE EC/LE (LT D SULLIVAN) MONTEREY CA 93943	1 1 1 1
JOHNS HOPKINS UNIVERSITY ATTN BRUCE BLUM JOHNS HOPKINS ROAD LAUREL MD 20723-6099	1	NAVAL RESEARCH LABORATORY ATTN RALPH D JEFFORDS BRUCE LABAW CODE 5543 (C MEADOWS) 4557 OVERLOOK AVE WASHINGTON DC 20375	1 1 1
JRS RESEARCH LABORATORIES ATTN ERWIN WARSHAWSKY 1036 WEST TAFT AVENUE ORANGE CA 92665	1		
NRAD ATTN CODE 41 (G MYERS) SAN DIEGO CA 92152	1		

DISTRIBUTION (CONT.)

	<u>Copies</u>		<u>Copies</u>
NUWC		TEXAS A&M UNIVERSITY	
ATTN JOHN RUMBUT JR	1	ATTN SWAMINATHAN NATARAJAN	1
BLDG 1171-3		COLLEGE STATION TX 77843	
NEWPORT RI 02841-2047			
NUWC DET	1	TRIDENT SYSTEMS INC	
ATTN THOMAS C CHOINSKI	1	ATTN NICHOLAS KARANGELLEN	1
JAMES B HALL JR	1	LAURA HINTON	1
STEVE HARRISON	1	10201 LEE HWY SUITE 300	
JOHN THANOS	1	FAIRFAX VA 22030	
NEW LONDON CT 06320			
NJ INSTITUTE OF TECHNOLOGY		UNIVERSITY OF CALIFORNIA IRVINE	
ATTN ALEXANDER STOYENKO	1	ATTN KANE KIM	1
LONNIE R WELCH	1	IRVINE CA 92717	
UNIVERSITY HEIGHTS		UNIVERSITY OF CALIFORNIA	
NEWARK NJ 07102		SAN DIEGO	
		ATTN FLAVIU CRISTIAN	1
OFFICE OF CHIEF OF NAVAL RESEARCH		LA JOLLA CA 92093-0114	
ATTN JAMES G SMITH	1	UNIVERSITY OF MARYLAND	
800 NORTH QUINCY STREET		ATTN CHRIS BIOW	1
ARLINGTON VA 22217-5009		MANASSA SAKSANA	1
PLANNING CONSULTANTS INC		SCOTT KING WALKER	1
ATTN ROBERT O'NEILL	1	COLLEGE PARK MD 20742	
PO BOX 1676		UNIVERSITY OF NEBRASKA	
DAHLGREN VA 22448		LINCOLN	
SOFTWARE PRODUCTIVITY		ATTN ROGER M KIECKHAFFER	1
CONSORTIUM		FERGUSON HALL	
ATTN PRASAD CHINTAMANENI	1	LINCOLN NE 68588-0115	
2214 ROCKHILL ROAD		UNIVERSITY OF PENNSYLVANIA	
HERNDON VA 22070		ATTN INSUP LEE	1
STA INC		200 SOUTH 33RD ST	
ATTN MARY E BLANCHARD	1	PHILADELPHIA PA 19104-6389	
4001 NORTH FAIRFAX DRIVE		UNIVERSITY OF RHODE ISLAND	
ARLINGTON VA 22302		ATTN VICTOR WOLFE	1
TASC		DEPT OF COMPUTER SCIENCE URI	
ATTN CHARLES M KOPLIK	1	KINGSTON RI 02881	
55 WALKERS BROOK DRIVE		UNIVIEW SYSTEMS	
READING MA 01867		ATTN ARMEN GABRIELIAN	1
		1192 ELENA PRIVADA	
		MOUNTAIN VIEW CA 94040	

DISTRIBUTION (CONT.)

	<u>Copies</u>			<u>Copies</u>
VPI&SU		K52	(W FARR)	1
ATTN OSMAN BALCI	1	N14	(M WILSON)	1
DEPARTMENT OF COMPUTER SCIENCE		N30	(H CRISP)	1
BLACKSBURG VA 24061		R33	(H SZU)	1
		U25	(D BERGSTEIN)	1
VPI&SU		U25	(PAUL CRAUN)	1
ATTN RICHARD NANCE	1	U25	(PHILIP CRAUN)	1
320 FEMOYER HALL		U25	(S LE)	1
BLACKSBURG VA 24061		U302	(P HWANG)	5
		U33	(D CHOI)	2
INTERNAL DISTRIBUTION		U33	(M EDWARDS)	2
C07 (B DUREN)	1	U33	(N HOANG)	2
D4 (M LACEY)	1	U33	(S HOWELL)	10
E52 (R LARSON)	1	U33	(M JENKINS)	2
G42 (C YEH)	1	U33	(C NGUYEN)	5
K43 (D ELLIOTT)	1	U33	(T PARK)	1
K51 (T WOOLFREY)	1	U33	(H ROTH)	1

REPORT DOCUMENTATION PAGEForm Approved
OMB No 0704 0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE 20-24 July 1992	3. REPORT TYPE AND DATES COVERED
4. TITLE AND SUBTITLE Proceedings of the 1992 Complex Systems Engineering Synthesis and Assessment Technology Workshop		5. FUNDING NUMBERS	
6. AUTHOR(S) Cuong M. Nguyen, Coordinator		8. PERFORMING ORGANIZATION REPORT NUMBER NSWCDD/MP-92/304	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Surface Warfare Center Dahlgren Division White Oak Detachment 10901 New Hampshire Avenue Silver Spring, MD 20903-5000		10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.		12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The emphasis of CSESAW '92 is on exploring system-level design synthesis and assessment capabilities for mission critical computer systems. These capabilities will facilitate the development of such systems from informal system requirements, through the design phase prototyping, and into implementation and post deployment. Component products produced by these capabilities are specifications that subenvironments will receive. The focus of this workshop is the development and integration of these multiple technologies and the exploration of the creation of a system-level engineering discipline with support technologies to provide potential high payoff solutions to the difficult problems encountered by designers, developers, and maintainers of large, complex, real-time systems. The emphasis is on resolving system-level technology issues that cut across component boundaries, such as those associated with system behavior requirements of real time, fault tolerance, and security.			
14. SUBJECT TERMS Systems engineering Design capture and analysis Massively Inter- System design synthesis Security connected model Requirements and traceability Design optimization Dependability			15. NUMBER OF PAGES 618
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAR